INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

APLICACIONES PARA COMUNICACIONES EN RED

UNIDAD I : Sockets de flujo

PRÁCTICA 1 – SERVICIO DE TRANSFERENCIA DE ARCHIVOS – TIENDA EN LÍNEA

Grupo 6CM2 - Ciclo 2026A

Alumnas:

Cruz Rodríguez Arely Amairani

Ortiz Villaseñor Alexandra

Fecha de entrega: Martes 7 de octubre de 2025

INTRODUCCIÓN

En las comunicaciones en red, el intercambio de información entre distintos equipos requiere establecer una conexión confiable que permita enviar y recibir datos de forma segura. Para lograrlo, se utiliza el modelo Cliente-Servidor, un esquema de comunicación en el cual un proceso actúa como servidor, ofreciendo servicios o recursos, y otro como cliente, solicitándolos. El servidor se mantiene a la espera de peticiones en un puerto específico, mientras que el cliente inicia la conexión hacia dicho puerto. Una vez establecida la conexión, ambos procesos pueden comunicarse intercambiando información de manera continua y controlada.

En esta práctica, se hace uso de sockets de flujo, también llamados *stream sockets*, los cuales utilizan el Protocolo de Control de Transmisión (TCP). Este tipo de socket garantiza que los datos se envíen de forma ordenada, sin pérdidas ni duplicaciones, gracias a las características propias de TCP, como el control de errores, la retransmisión de paquetes y el manejo del flujo de datos. Por esta razón, los sockets de flujo son ideales para la transferencia de archivos, donde es indispensable que los datos lleguen completos y en el mismo orden en que fueron enviados.

Un aspecto clave en este proceso es la serialización, que consiste en convertir los datos, ya sean estructuras, objetos o archivos binarios en una secuencia de bytes lista para ser transmitida a través del socket. Este procedimiento permite que la información mantenga su estructura y pueda ser reconstruida correctamente del lado del servidor (deserialización). En la práctica de transferencia de archivos, la serialización garantiza que cada fragmento del archivo se envíe de manera legible para el sistema receptor, sin pérdida de contenido ni errores de interpretación, facilitando así una comunicación robusta y confiable.

La aplicación desarrollada implementa un servicio de transferencia de archivos mediante un enfoque cliente-servidor, en el que el servidor se encarga de recibir y almacenar los archivos enviados por uno o varios clientes. Cada cliente establece una conexión TCP con el servidor, serializa los datos del archivo y los envía a través del socket de flujo, asegurando que la información llegue íntegra y en orden.

De esta forma, la práctica permite comprender de manera integral cómo los sockets de flujo, el modelo cliente-servidor y la serialización de datos interactúan para crear un sistema de comunicación confiable entre dispositivos, sentando las bases para el desarrollo de aplicaciones distribuidas más complejas en el ámbito de las redes de computadoras.

DESARROLLO

CLASE CARRITO DE COMPRA

Comenzamos definiendo la clase Cart, que representa un carrito de compras simple que almacena pares de identificadores de artículos (itemId) con sus respectivas cantidades ordenadas, mediante un HashMap. Todos sus métodos están sincronizados para garantizar la seguridad en entornos con múltiples hilos, evitando accesos concurrentes no controlados.

La clase permite agregar artículos sumando cantidades con el método add, establecer una cantidad exacta o eliminar el artículo si la cantidad es cero mediante setQuantity, eliminar directamente con remove, obtener una copia del contenido con getItems y verificar si el carrito está vacío con isEmpty. Este diseño ofrece un manejo eficiente y seguro del inventario dentro de una sesión, protegiendo la integridad de los datos frente a modificaciones simultáneas.

Cart.java

```
import java.util.HashMap;
import java.util.Map;

public class Cart {
    // itemId -> cantidad
    private final Map<Integer, Integer> items = new HashMap<>();

    public synchronized void add(int itemId, int qty) {
        items.merge(itemId, qty, Integer::sum);
    }

    public synchronized void setQuantity(int itemId, int qty) {
        if (qty <= 0) items.remove(itemId);
        else items.put(itemId, qty);
    }

    public synchronized void remove(int itemId) {
        items.remove(itemId);
    }

    public synchronized Map<Integer, Integer> getItems() {
        return new HashMap<>(items);
    }

    public synchronized boolean isEmpty() {
        return items.isEmpty();
    }
}
```

CLASE ARTICULO

La clase Item modela un producto del inventario y es Serializable para poder enviarse o persistirse fácilmente; define atributos inmutables para la identidad y descripción (id, tipo, nombre, marca) y el precio, además de un campo mutable existencia que representa el stock.

Proporciona getters para consultar sus datos y métodos sincronizados para manipular el inventario de forma segura en entornos concurrentes: getExistencia() devuelve el stock actual, decrementar(int cantidad) valida que la cantidad sea positiva y que haya suficiente existencia antes de restar (retorna true si se concreta y false en caso contrario), e incrementar(int cantidad) aumenta el stock; el uso de synchronized evita condiciones de carrera cuando varios hilos actualizan el mismo objeto.

El método toString() ofrece una representación legible del artículo con formato "ID:… | tipo – nombre | $precio | stock:…", útil para logs o interfaces de texto.

Item.java

```
import java.io.Serializable;

public class Item implements Serializable {
    private final int id;
    private final String tipo; // categoría o tipo
    private final String nombre;
    private final double precio;
    private final String marca;
    private int existencia;

    public Item(int id, String tipo, String nombre, double precio, String marca, int existencia) {
        this.id = id;
        this.tipo = tipo;
        this.nombre = nombre;
        this.precio = precio;
        this.marca = marca;
        this.existencia = existencia;
    }

    public int getId() { return id; }
    public String getTipo() { return tipo; }
    public String getNombre() { return nombre; }
    public double getPrecio() { return precio; }
    public String getMarca() { return marca; }
    public synchronized int getExistencia() { return existencia; }

    public synchronized boolean decrementar(int cantidad) {
        if (cantidad <= 0) return false;
        if (existencia >= cantidad) {
            existencia -= cantidad;
            return true;
        }
        return false;
    }
    public synchronized void incrementar(int cantidad) {
        existencia += cantidad;
    }

    @Override
    public String toString() {
        return String.format("ID:%d | %s | %s | %s | $%.2f | stock:%d", id, nombre, marca, tipo, precio, existencia);
    }
}
```

CLASE SERVIDOR

La clase Server implementa un servidor TCP concurrente que escucha en un puerto y atiende múltiples clientes en paralelo usando un ExecutorService con newCachedThreadPool, una clase predefinida de Java que administra la creación de hilos reutilizables; mantiene un inventario en memoria con ConcurrentHashMap<Integer, Item> (id → Item) inicializado en seedInventory() la lista de artículos que ofrece nuestra tienda en línea.

Por cada conexión, envía "WELCOME" al cliente y procesa líneas de texto con un protocolo simple: SHOW_ALL devuelve todos los artículos, LIST_TIPO <tipo> filtra por tipo, SEARCH <query> busca un artículo por su marca o nombre, CHECK <id> responde OK <existencia>, y FINALIZAR <usuario> <id:qty,...> valida y descuenta existencias de forma atómica dentro de un bloqueo global compraLock (para evitar condiciones de carrera), calcula el total y devuelve un "ticket" textual (Ticket) si todo es correcto; ante errores de formato, id inexistente o stock insuficiente, responde ERROR.

La lectura/escritura se hace con BufferedReader/PrintWriter, y cada cliente se gestiona en un hilo independiente mediante handleClient, que parsea y enruta los comandos en processCommand, asegurando un flujo claro y seguro para consultar stock, listar productos y cerrar compras contra el inventario compartido.

Server.java

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.*;
import java.util.concurrent.*;

public class Server {
    private final int port;
    private final ConcurrentHashMap<Integer, Item> inventory = new ConcurrentHashMap<>();
    private final ExecutorService pool = Executors.newCachedThreadPool();

    //Constructor de la clase Server
    public Server(int port) {
        this.port = port; //Puerto de conexion
        seedInventory(); //Se llena la estrctura inventory
    }

    //Llenado del hash Map con los datos del inventario
    private void seedInventory() {
       inventory.put(1, new Item(1, "Electronicos", "Audifonos XYZ", 499.0, "JBL", 10));
      inventory.put(2, new Item(2, "Electronicos", "Cargador USB-C", 199.0, "TechFast", 15));
      inventory.put(3, new Item(3, "Ropa", "Playera Azul", 299.0, "WearIt\t", 8));
      inventory.put(4, new Item(4, "Hogar", "Taza Ceramica", 89.0, "DecoHome", 25));
      inventory.put(5, new Item(5, "Electronicos",    "Smartwatch Deportivo", 1599.0, "Chronos",  5));
      inventory.put(6, new Item(6,    "Juguetes", "Set de Bloques Armables",  350.0,  "Blocky",   12));
      inventory.put(7, new Item(7,    "Ropa", "Jeans Slim Fit Negro", 799.0,  "DenimCo",  7));
      inventory.put(8, new Item(8,    "Hogar",    "Veladora Aromática Vainilla",  120.0,  "Scents",   30));
      inventory.put(9, new Item(9, "Libros",  "Novela de Misterio 'La Clave'",    390.0,  "Planeta",  20));
      inventory.put(10, new Item(10, "Electronicos", "Mouse Inalámbrico Ergonómico",  320.0,  "Periph",   18));
      inventory.put(11, new Item(11, "Deportes", "Tapete de Yoga Antideslizante", 450.0,  "FitLife",  14));
      inventory.put(12, new Item(12, "Hogar", "Set de 3 Cuchillos de Chef",   850.0,  "CutPro",   9));
      inventory.put(13, new Item(13, "Ropa", "Sudadera con Capucha Gris",     650.0,  "Cozy", 11));
      inventory.put(14, new Item(14, "Libros", "Libro de Recetas Italianas",  410.0,  "GourmetPub",   6));
      inventory.put(15, new Item(15, "Electronicos", "Webcam Full HD",    550.0,  "ZoomTech", 16));
      inventory.put(16, new Item(16, "Belleza", "Crema Hidratante Facial",    280.0,  "Natura",   22));
      inventory.put(17, new Item(17, "Hogar", "Lámpara de Escritorio LED",    599.0,  "LightUp",  13));
      inventory.put(18, new Item(18, "Juguetes", "Drone Pequeño para Niños",  999.0,  "FlyFun",   4));
      inventory.put(19, new Item(19, "Deportes", "Botella de Agua Térmica 1L",    250.0,  "Hydro",    28));
      inventory.put(20, new Item(20, "Belleza", "Set de Brochas de Maquillaje",   399.0,  "BrushKit", 17));
    }
```

```java
//Metodo start de la clase server
public void start() throws IOException {
   try (ServerSocket serverSocket = new ServerSocket(port)) { // se crea el server Socket
       System.out.println("Servidor escuchando en puerto " + port);
       while (true) {
           Socket client = serverSocket.accept(); //En cuanto llegue una peticion de un cliente, esta se
hacepta y se define un socket para esta conexion
           pool.submit(() -> handleClient(client)); //Se inicializa un hilo para ejecutar el metodo handle Client
       }
   } finally {
   pool.shutdown(); //Reutilizar hilo
   }
}
//Metodo handle Client
private void handleClient(Socket socket) {
   String clientInfo = socket.getRemoteSocketAddress().toString();
   System.out.println("Conexión: " + clientInfo);
   try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
       PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
         String line;
         out.println("WELCOME"); // saludo
         while ((line = in.readLine()) != null) {
            String resp = processCommand(line.trim());
            out.println(resp);
         }
   } catch (Exception e) {
        System.err.println("Error cliente: " + e.getMessage());
   } finally {
        try { socket.close(); } catch (IOException ignored) {}
        System.out.println("Desconectado: " + clientInfo);
   }
}
private String processCommand(String line) {
    if (line.isEmpty()) return "ERROR Empty command";
    String[] parts = line.split(" ", 2);
    String cmd = parts[0].toUpperCase();
    String arg = parts.length > 1 ? parts[1].trim() : "";
       switch (cmd) {
           case "SHOW_ALL":
               return showAll();
            case "LIST_TIPO":
               return listTipo(arg);
            case "SEARCH":
               return searchByNameOrBrand(arg);
            case "CHECK":
               return checkItem(arg);
            case "FINALIZAR":
               return finalizarCompra(arg);
            default:
               return "ERROR Unknown command";
        }
    }

    private String showAll() {
        StringBuilder sb = new StringBuilder();
        inventory.values().forEach(i -> sb.append(i.toString()).append("\n"));
        return "OK\n" + sb.toString();
    }
```

```
    private String listTipo(String tipo) {
        if (tipo.isEmpty()) return "ERROR Tipo vacío";
        StringBuilder sb = new StringBuilder();
        inventory.values().stream()
                .filter(i -> i.getTipo().equalsIgnoreCase(tipo))
                .forEach(i -> sb.append(i.toString()).append("\n"));
        return "OK\n" + sb.toString();
    }

    private String searchByNameOrBrand(String query) {
    if (query.isEmpty()) return "ERROR Consulta vacía";
    StringBuilder sb = new StringBuilder();
    String q = query.toLowerCase();
    inventory.values().stream()
            .filter(i -> i.getNombre().toLowerCase().contains(q) ||
i.getMarca().toLowerCase().contains(q))
            .forEach(i -> sb.append(i.toString()).append("\n"));
    return sb.length() > 0 ? "OK\n" + sb.toString() : "ERROR No se encontraron coincidencias";
}

    private String checkItem(String arg) {
        try {
            int id = Integer.parseInt(arg);
            Item it = inventory.get(id);
            if (it == null) return "ERROR Item no encontrado";
            return "OK " + it.getExistencia();
        } catch (NumberFormatException e) {
            return "ERROR id inválido";
        }
    }

    private final Object compraLock = new Object();

    private String finalizarCompra(String arg) {
        if (arg.isEmpty()) return "ERROR formato FINALIZAR <user> <items>";
        String[] parts = arg.split(" ", 2);
        if (parts.length < 2) return "ERROR formato FINALIZAR <user> <items>";
        String user = parts[0];
        String itemsStr = parts[1];

        Map<Integer, Integer> deseos = new HashMap<>();
        try {
            String[] pares = itemsStr.split(",");
            for (String p : pares) {
                String[] kv = p.split(":");
                int id = Integer.parseInt(kv[0].trim());
                int qty = Integer.parseInt(kv[1].trim());
                if (qty <= 0) return "ERROR cantidad inválida para id " + id;
                deseos.put(id, qty);
            }
        } catch (Exception e) {
            return "ERROR formato items inválido. Ej: 1:2,3:1";
        }

        // bloqueo global de compra para validar y decrementar atomícamente
        synchronized (compraLock) {
            // validar existencias
            for (Map.Entry<Integer, Integer> e : deseos.entrySet()) {
                Item it = inventory.get(e.getKey());
                if (it == null) return "ERROR item " + e.getKey() + " no existe";
                if (it.getExistencia() < e.getValue()) {
```

```
                    return "ERROR no hay suficiente stock para item " + e.getKey() + ".
disponible=" + it.getExistencia();
                }
            }
            // decrementar
            double total = 0.0;
            for (Map.Entry<Integer, Integer> e : deseos.entrySet()) {
                Item it = inventory.get(e.getKey());
                boolean ok = it.decrementar(e.getValue());
                if (!ok) {
                    // rollback parcial: para simplicidad incrementamos lo que ya se decrementó
                    // (pero con el lock global esto no debería pasar)
                    for (Map.Entry<Integer, Integer> rolled : deseos.entrySet()) {
                        if (rolled.getKey() == e.getKey()) break;
                        inventory.get(rolled.getKey()).incrementar(rolled.getValue());
                    }
                    return "ERROR al decrementar item " + e.getKey();
                }
                total += it.getPrecio() * e.getValue();
            }
            // generar ticket y devolverlo (como texto)
            Ticket ticket = new Ticket(user, deseos, total);
            return "OK\n" + ticket.toString();
        }
    }

    //Instancia de la clase Server
    public static void main(String[] args) throws IOException {
        int port = 5555;
        Server s = new Server(port);
        s.start();
    }
}
```

CLASE CLIENTE

ClientCLI es un cliente de consola que se conecta a un servidor TCP (Socket) en host:port, intercambiando mensajes de texto mediante BufferedReader y PrintWriter para operar un catálogo/venta con un protocolo simple por comandos: muestra un menú y, según la opción, envía SHOW_ALL (listar todo), LIST_TIPO <tipo> (filtrar por tipo), CHECK <id>(consultar stock) o FINALIZAR <usuario> <id:qty,...>; localmente mantiene un carrito (Cart) donde agrega solo si el servidor confirma existencias, además permite editar cantidades o eliminar ítems con editCart().

La lectura de respuestas se hace en printResponse, que acumula líneas hasta encontrar una línea en blanco o cuando serverIn.ready() indica que no hay más datos, asumiendo que el servidor delimita cada respuesta con una línea vacía. Ojo: tras finalizar la compra el código intenta vaciar el carrito con cart.getItems().clear(), pero getItems()devuelve una **copia defensiva**, así que ese clear() no afecta al carrito real; habría que añadir un método clear() en Cart o reasignar cantidades a 0 con setQuantity para vaciarlo correctamente.

ClientCLI.java

```
import java.io.*;
import java.net.Socket;
import java.util.Map;
import java.util.Scanner;
import java.util.stream.Collectors;

public class ClientCLI {
    private final String host;
    private final int port;
    private final Cart cart = new Cart();
    private final Scanner sc = new Scanner(System.in);

    //Constructor de la clase Client
    public ClientCLI(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void start() {
     try (Socket socket = new Socket(host, port);
      BufferedReader serverIn = new BufferedReader(new InputStreamReader(socket.getInputStream()));
             PrintWriter serverOut = new PrintWriter(socket.getOutputStream(), true)) {

      System.out.println(serverIn.readLine()); // WELCOME
      boolean running = true;
      while (running) {
           showMenu();
           String opt = sc.nextLine().trim();
           switch (opt) {
               case "1":
                   serverOut.println("SHOW_ALL");
                   printResponse(serverIn);
               break;
               case "2":
                   System.out.print("Tipo: ");
                   String tipo = sc.nextLine().trim();
                   serverOut.println("LIST_TIPO " + tipo);
                   printResponse(serverIn);
               break;
```

```
         case "3":
             System.out.print("Busqueda: ");
             String query = sc.nextLine().trim();
             serverOut.println("SEARCH " + query);
             printResponse(serverIn);
         break;
         case "4":
             System.out.print("ID a agregar: ");
             int id = Integer.parseInt(sc.nextLine().trim());
             System.out.print("Cantidad: ");
             int qty = Integer.parseInt(sc.nextLine().trim());
             // preguntar al servidor existencia actual antes de agregar
             serverOut.println("CHECK " + id);
             String check = serverIn.readLine();
             if (check.startsWith("OK")) {
                int disponibles = Integer.parseInt(check.split(" ")[1]);
                if (disponibles >= qty) {
                     cart.add(id, qty);
                     System.out.println("Agregado al carrito.");
                } else {
                   System.out.println("No hay suficientes existencias. disponibles=" + disponibles);
                    }
             } else {
                System.out.println("Error al checar item: " + check);
             }
         break;
         case "5":
             editCart();
         break;
         case "6":
             if (cart.isEmpty()) {
                System.out.println("Carrito vacío.");
                break;
             }
             System.out.print("Nombre de usuario para ticket: ");
             String user = sc.nextLine().trim();
             String items = cart.getItems().entrySet().stream()
                        .map(e -> e.getKey() + ":" + e.getValue())
                        .collect(Collectors.joining(","));
             serverOut.println("FINALIZAR " + user + " " + items);
             printResponse(serverIn);
             // si OK -> vaciar carrito localmente (asumido comprado)
              // String last = serverIn.readLine(); // already read in printResponse? careful <-
sí ya se lee en printResponse
              // NOTE: printResponse reads one line and prints additional lines only if OK\ntext
              // To keep simple, re-request final response:
              // Actually modify printResponse to return the full response. For brevity, assume
server sent full.
              cart.getItems().clear();
         break;
         case "0":
              running = false;
         break;
         default:
              System.out.println("Opción inválida");
      }
   }
   System.out.println("Saliendo cliente.");
 } catch (Exception e) {
         System.err.println("Error cliente: " + e.getMessage());
         e.printStackTrace();
```

```
        }
    }

    private void editCart() {
        while (true) {
            System.out.println("Carrito:");
            cart.getItems().forEach((k,v) -> System.out.println("ID:" + k + " x " + v));
            System.out.println("a) Cambiar cantidad");
            System.out.println("b) Eliminar item");
            System.out.println("c) Volver");
            String opt = sc.nextLine().trim();
            if (opt.equalsIgnoreCase("a")) {
                System.out.print("ID: ");
                int id = Integer.parseInt(sc.nextLine().trim());
                System.out.print("Nueva cantidad: ");
                int q = Integer.parseInt(sc.nextLine().trim());
                cart.setQuantity(id, q);
            } else if (opt.equalsIgnoreCase("b")) {
                System.out.print("ID a eliminar: ");
                int id = Integer.parseInt(sc.nextLine().trim());
                cart.remove(id);
            } else break;
        }
    }

    private void showMenu() {
        System.out.println("\n--- MENU ---");
        System.out.println("1) Mostrar todos los artículos");
        System.out.println("2) Buscar por tipo");
        System.out.println("3) Buscar por marca o nombre");
        System.out.println("4) Agregar al carrito (valida stock antes)");
        System.out.println("5) Editar carrito");
        System.out.println("6) Finalizar compra y obtener ticket");
        System.out.println("0) Salir");
        System.out.print("Opcion: ");
    }

    private void printResponse(BufferedReader serverIn) throws IOException {
        StringBuilder response = new StringBuilder();

        while (true) {
            String line = serverIn.readLine();
            if (line == null) break;        // servidor cerró conexión
            if (line.isBlank()) break;      // línea vacía = fin de respuesta
            response.append(line).append("\n");
            // si el servidor no envía línea vacía, esto leerá solo la primera y sale
            if (!serverIn.ready()) break;
        }

        System.out.println(response.toString());
    }

    public static void main(String[] args) {
        ClientCLI c = new ClientCLI("localhost", 5555);
        c.start();
    }
}
```

CLASE TICKET DE COMPRA

La clase Ticket modela el comprobante de una compra: guarda el usuario (user), el mapa de artículos con cantidades (detalles donde la llave es el ID y el valor la cantidad), el monto total y la fecha de emisión capturada al construir el objeto con LocalDateTime.now().

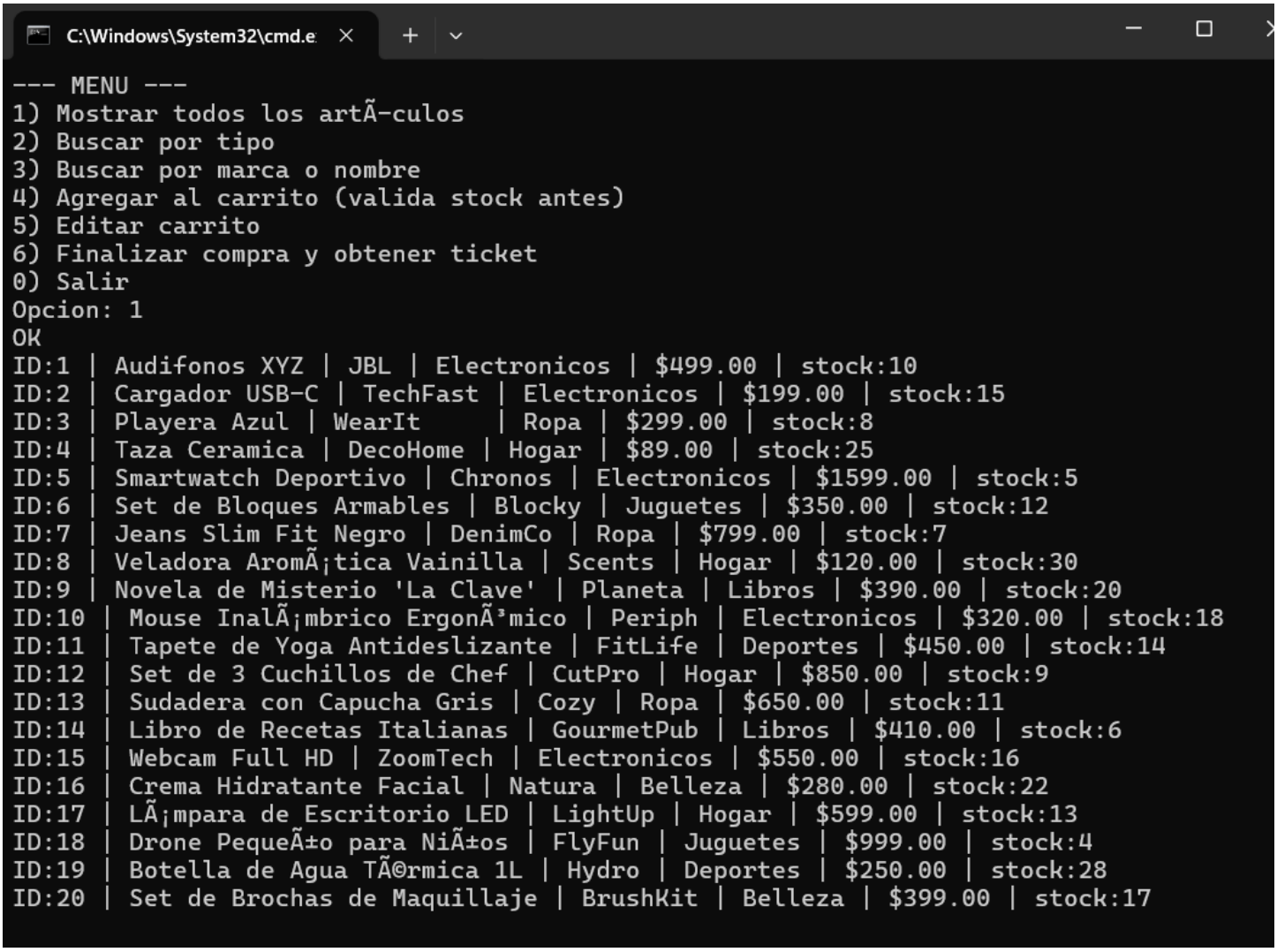
Sus campos son final para mantenerlos inmutables tras la creación. El método toString() formatea el ticket en texto legible: imprime encabezado, usuario, fecha con patrón yyyy-MM-dd HH:mm:ss, lista cada par Item ID: <id> x <cantidad> recorriendo el mapa, y cierra con el total alineado a dos decimales mediante String.format. Con esto se obtiene una representación clara y trazable del comprobante para enviar por red o registrar en logs.

Ticket.java

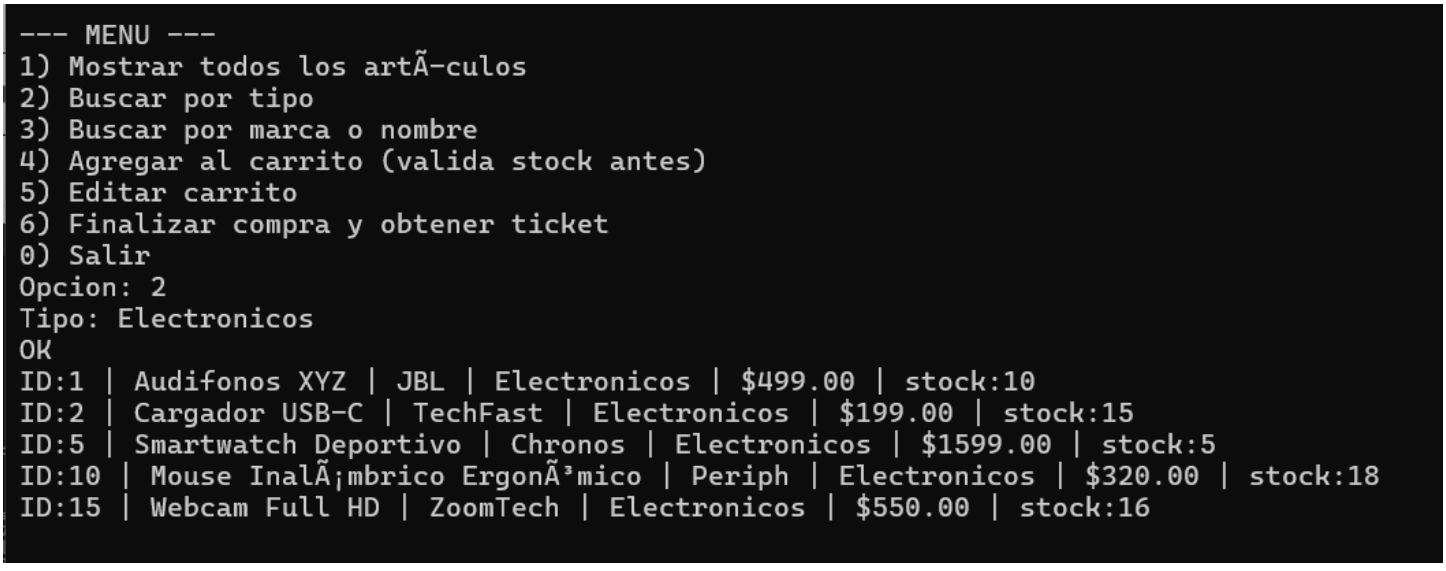
```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Map;

public class Ticket {
    private final String user;
    private final Map<Integer, Integer> detalles;
    private final double total;
    private final LocalDateTime fecha;

    public Ticket(String user, Map<Integer, Integer> detalles, double total) {
        this.user = user;
        this.detalles = detalles;
        this.total = total;
        this.fecha = LocalDateTime.now();
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("TICKET\n");
        sb.append("Usuario: ").append(user).append("\n");
        sb.append("Fecha: ").append(fecha.format(DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss"))).append("\n");
        sb.append("Detalles:\n");
        detalles.forEach((id, qty) -> sb.append("  Item ID: ").append(id).append(" x
").append(qty).append("\n"));
        sb.append(String.format("TOTAL: $%.2f\n", total));
        return sb.toString();
    }
}
```

PRUEBAS y REQUERIMIENTOS

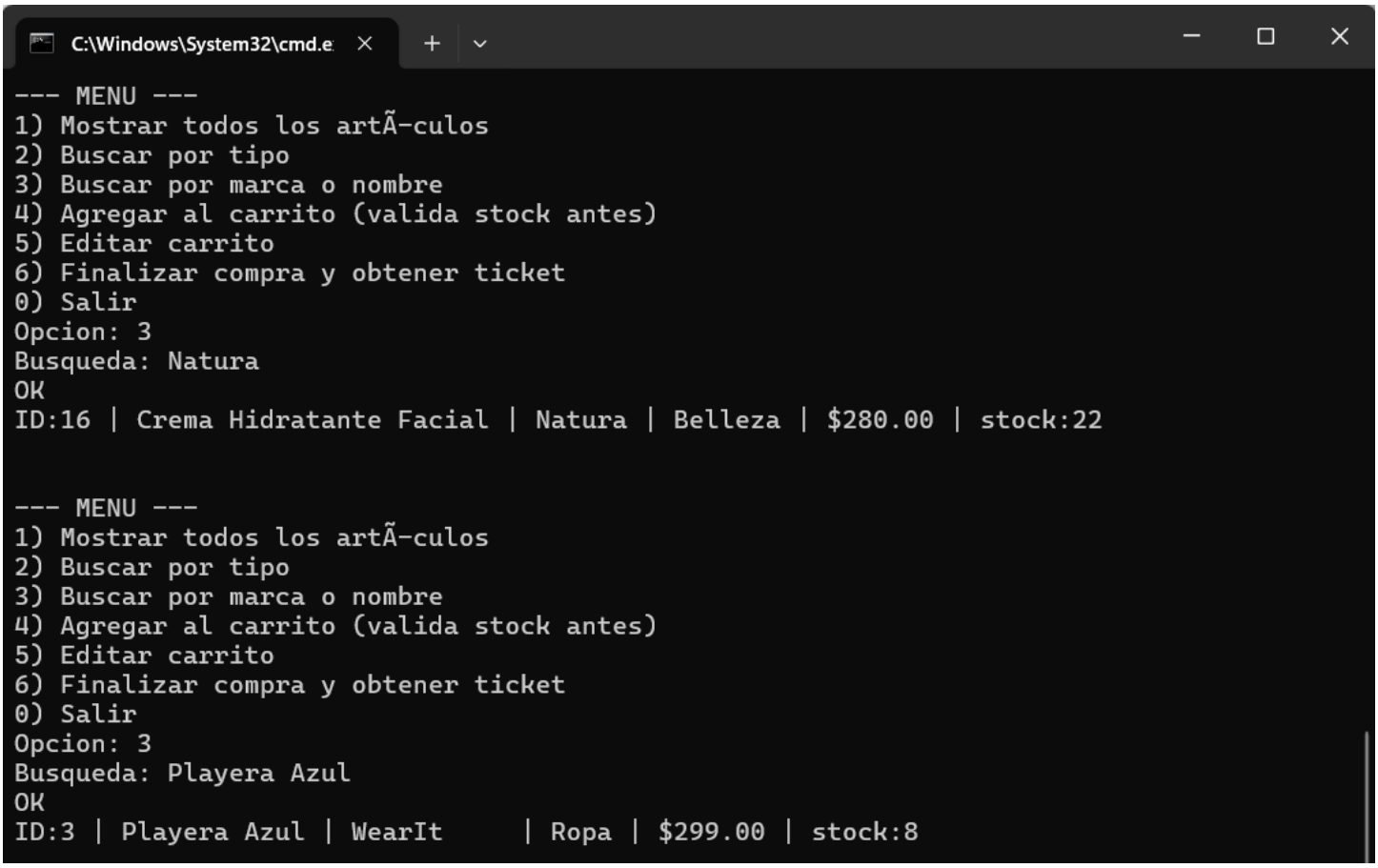
Visualización de los artículos de la tienda



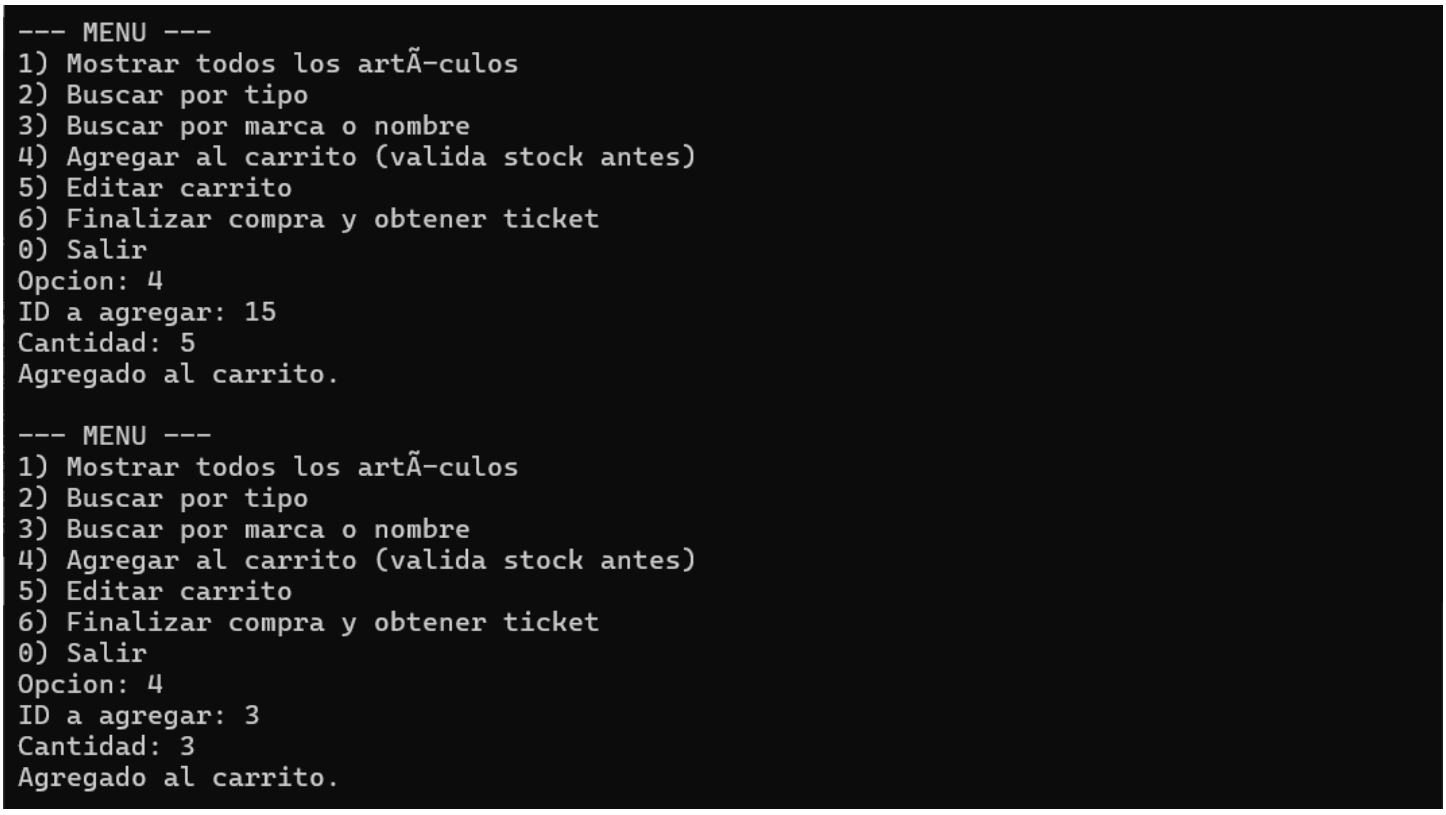
Búsqueda por tipo de producto:



Búsqueda por marca o nombre de un producto:



Agregar producto al carrito de compras



Editar carrito de compras

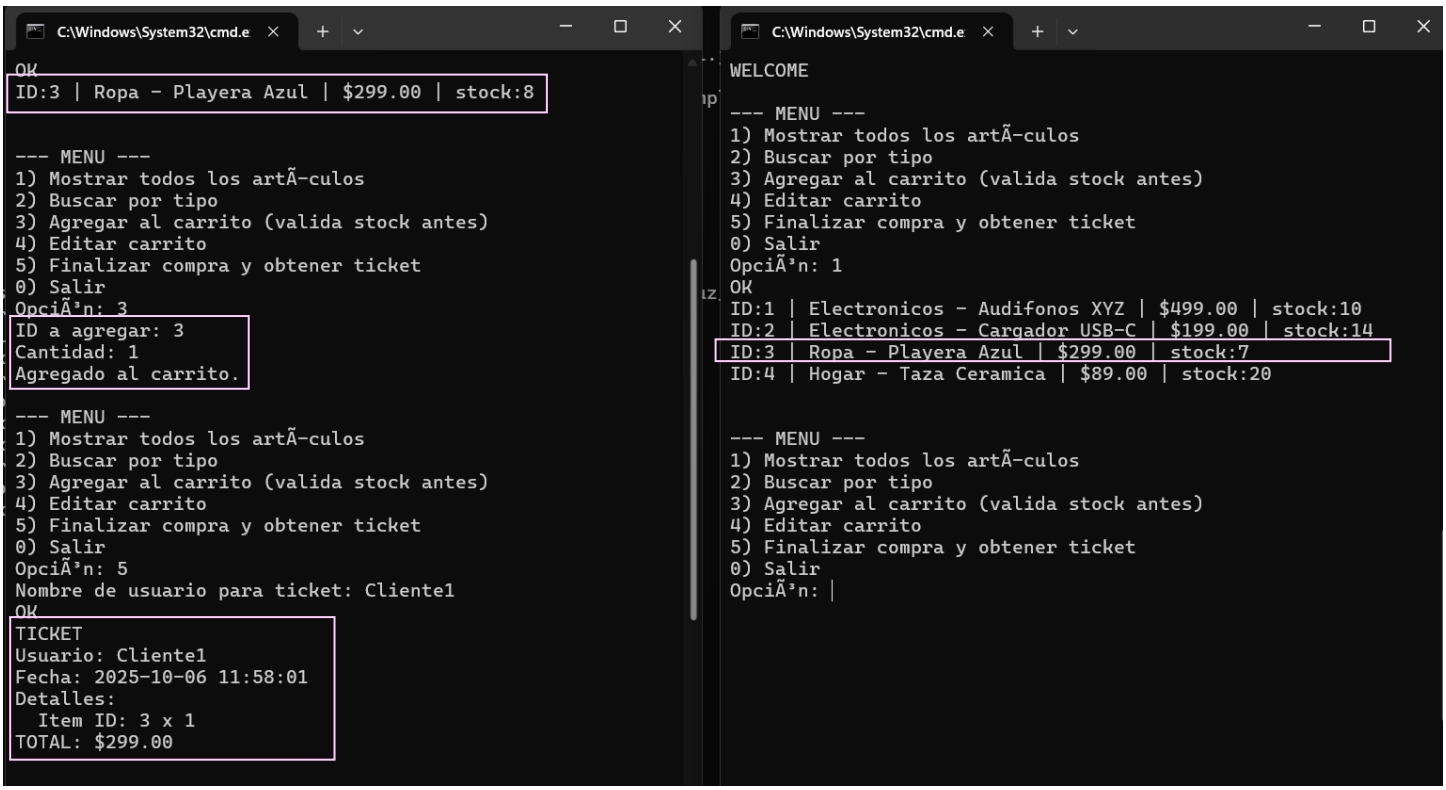
```
--- MENU ---
1) Mostrar todos los artÃ-culos
2) Buscar por tipo
3) Buscar por marca o nombre
4) Agregar al carrito (valida stock antes)
5) Editar carrito
6) Finalizar compra y obtener ticket
0) Salir
Opcion: 5
Carrito:
ID:3 x 3
ID:15 x 5
a) Cambiar cantidad
b) Eliminar item
c) Volver
a
ID: 15
Nueva cantidad: 2
Carrito:
ID:3 x 3
ID:15 x 2
a) Cambiar cantidad
b) Eliminar item
c) Volver
b
ID a eliminar: 3
Carrito:
ID:15 x 2
a) Cambiar cantidad
b) Eliminar item
c) Volver
c
```

Finalizar compra y generar ticket

```
--- MENU ---
1) Mostrar todos los artÃ-culos
2) Buscar por tipo
3) Buscar por marca o nombre
4) Agregar al carrito (valida stock antes)
5) Editar carrito
6) Finalizar compra y obtener ticket
0) Salir
Opcion: 6
Nombre de usuario para ticket: Cliente1
OK
TICKET
Usuario: Cliente1
Fecha: 2025-10-06 20:56:14
Detalles:
  Item ID: 15 x 2
TOTAL: $1100.00
```

Actualización del stock en las compras:

Corriendo en simultaneo 2 clientes (terminales)



CONCLUSIONES

Cruz Rodríguez Arely Amairani

Esta práctica nos permitió poner en práctica la teoría vista sobre el modelo Cliente-Servidor y la importancia de los Sockets de Flujo (TCP). Logramos implementar una comunicación robusta donde se garantiza la entrega ordenada y sin pérdidas de datos. Un aprendizaje fundamental fue la necesidad de la serialización de la clase Item. Al implementar la interfaz Serializable, aseguramos que los objetos de inventario se pudieran convertir eficientemente en un flujo de bytes para su transmisión a través de la red, facilitando la interpretación y reconstrucción precisa de la información en el cliente. Este mecanismo es indispensable para construir sistemas de transferencia de datos fiables.

Ortiz Villaseñor Alexandra

La práctica fue clave para entender la implementación de un servidor TCP concurrente. Al utilizar un ExecutorService con un *pool* de hilos, logramos que el servidor pudiera atender a múltiples clientes en paralelo, asignando un nuevo hilo para la ejecución del método handleClient por cada conexión entrante. Este diseño de concurrencia garantiza la integridad del inventario centralizado y evita inconsistencias durante interacciones simultáneas.

FUENTES

[1] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. Upper Saddle River, NJ, USA: Pearson Education, 2011.

[2] Oracle, “Trail: Custom Networking and the Java Sockets API,” *The Java™ Tutorials*, Oracle, 2024. [Online]. Available: https://docs.oracle.com/javase/tutorial/networking/sockets/

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA, USA: Addison-Wesley, 1994.

[4] M. Pilgrim and S. Willison, *Dive Into Python 3*, New York, NY, USA: Apress, 2009. [Online]. Available: https://diveintopython3.net/