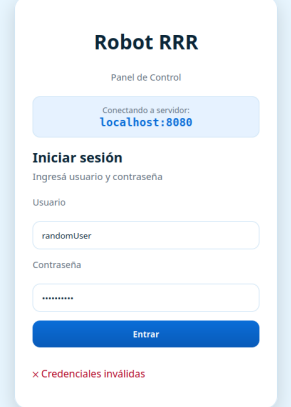
UNIVERSIDAD NACIONAL DE CUYO
Facultad de Ingeniería

Trabajo Práctico Integrador

Implementación Cliente/Servidor basado en protocolo de comunicación XML-RCP para control de un robot de 3 DoF tipo ”RRR”



CARRERA Ingeniería Mecatrónica

SEMESTRE Segundo Semestre 2025

CÁTEDRA Programación Orientada a Objetos

ACTIVIDAD Trabajo Práctico Integrador

ALUMNOS Federico Barrios Retta , 14101
Lucia Muñiz , 13934
Nicolas Patricelli , 14090

REPOSITORIO https://github.com/Ede118/Final-POO.git

FECHA noviembre 2025

Contents

1 Introducción

El presente trabajo se realiza en el marco del **Trabajo Práctico Integrador** de la asignatura **Programación Orientada a Objetos**, de la carrera **Ingeniería en Mecatrónica** de la Facultad de Ingeniería de la Universidad Nacional de Cuyo.

El objetivo general es **controlar de forma remota un robot de 3 grados de libertad (RRR) con efector final**, utilizando un enfoque fuertemente **orientado a objetos** y una arquitectura **cliente–servidor**:

- **Servidor** implementado en **C++**, responsable del modelo de dominio, la lógica de negocio y la comunicación con el simulador / firmware del robot mediante G-Code.
- **Cliente** implementado en **HTML + CSS + JavaScript**, responsable de ofrecer una interfaz de usuario accesible desde un navegador para operar el robot de forma local o remota.

El trabajo toma como base los requerimientos detallados en la consigna oficial, incluyendo:

- Separación clara entre **capa de modelo** (dominio del robot y sus comandos) y **capa de presentación**.
- Uso de **clases, herencia, agregación, polimorfismo, manejo de excepciones, contenedores** y gestión de recursos.
- Exposición de servicios del lado servidor mediante un mecanismo de **RPC** (específicamente XML-RPC) y manejo de **usuarios, permisos, logs y base de datos**.
- Implementación de una interfaz de usuario capaz de **enviar comandos G-Code**, cargar tareas desde archivos y monitorear el estado del robot.

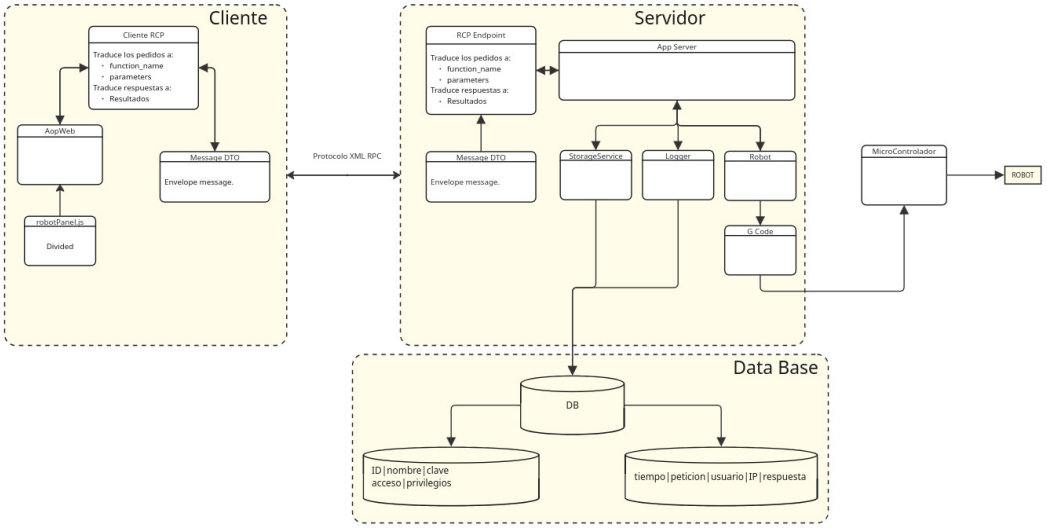
En este informe se describe la **arquitectura general** de la solución, los componentes principales del servidor en C++, el cliente web en JavaScript, la interfaz gráfica, detalles de instalación y ejecución, y finalmente se presentan comentarios, dificultades encontradas y posibles extensiones futuras.

2 Solución Propuesta

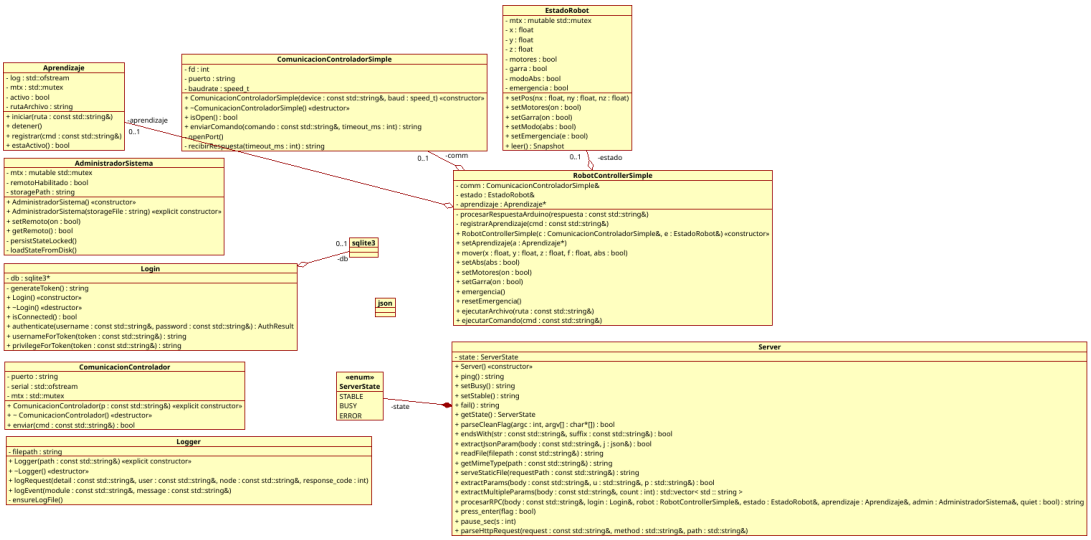
La solución implementada adopta un enfoque de **arquitectura por capas** y **modelo de dominio compartido**:

1. Una **capa de dominio** que modela:
   - El robot "RRR" y su espacio de trabajo.
   - Los comandos G-Code y mensajes intercambiados.
   - Los usuarios, credenciales y permisos.
   - El registro de eventos del sistema (log).
2. Una **capa de servicios** en el servidor C++ que:
   - Valida y procesa las peticiones de los clientes.
   - Se comunica por puerto serie con el simulador / firmware del robot.
   - Expone operaciones mediante un servidor RPC.
3. Una **capa de presentación** que:
   - Del lado servidor, ofrece una **CLI administrativa** para el usuario con perfil de administrador.
   - Del lado cliente, brinda una **interfaz web** para operación remota (HTML + CSS + JS).

A nivel conceptual:



Un diagrama de clases más fiel al proyecto sería:



El intercambio de información entre cliente y servidor se realiza a través de objetos de dominio serializables. De esta forma, tanto en C++ como en JavaScript se trabaja con el mismo concepto de "petición", favoreciendo la coherencia del diseño OO en "ambos extremos".

La siguiente sección detalla cómo se concreta esta arquitectura del lado del servidor C++.

2.1 Servidor con C++

El servidor está implementado en C++, organizando la lógica en clases y archivos que separan responsabilidades:

- **Módulo de dominio del robot**
  - Encapsula la lógica de control y estado:
  - Clase `EstadoRobot`: Modela el estado actual del robot (posición X, Y, Z; estado de motores y garra; modo absoluto/relativo; emergencia) de forma *thread-safe* para accesos concurrentes.

  - Clase `RobotControllerSimple`: Contiene la lógica de alto nivel. Recibe órdenes (como `mover` o `setMotores`) y las traduce a comandos G-Code.
- **Módulo de usuarios y seguridad**
  - Gestiona la autenticación y los permisos:
  - Clase `Login`: Centraliza toda la gestión de usuarios. Se conecta directamente a una base de datos **SQLite** (`users.sqlite3`) para crear la tabla y almacenar usuarios (username, password_hash, privilege).
  - El método `authenticate` valida las credenciales contra la BD. Si son exitosas, genera un token (`std::string`) y lo almacena temporalmente en un `static std::unordered_map` en memoria para validar las sesiones activas.
- **Módulo de logging y persistencia**
  - Responsable de mantener el archivo de log:
  - Clase `Logger`: Implementada como una instancia global (`logger`) que escribe eventos en un archivo.
  - El formato es **CSV** (como se ve en `logger.cpp`), y registra eventos (`logEvent`) y peticiones (`logRequest`) con *timestamp*, módulo y mensaje.
  - Es un log de tipo *append-only* (solo añade). El proyecto **no** implementa filtros ni reportes del lado del servidor; el log se consulta directamente o se exporta.
- **Módulo de comunicación con el robot (simulador)**
  - Implementa el enlace serie con el hardware:
  - Clase `ComunicacionControladorSimple`: Abstrae la comunicación por puerto serie (ej. `/dev/ttyUSB0`) usando `termios`.
  - Si el puerto físico falla al abrirse, la clase entra automáticamente en un **modo de simulación** que responde "SIM:OK" a los comandos.
  - El `RobotControllerSimple` usa esta clase para enviar G-Code (`G28`, `G1 X...`, `M3`, etc.) e interpretar las respuestas (`procesarRespuestaArduino`) para detectar `ok`, `error` o `alarm`.
- **Módulo RPC / red**
  - Expone la funcionalidad del servidor a los clientes remotos:
  - La lógica de red combina un *loop* de sockets C (`main.cpp`) con una clase `Server` (`server.cpp`).
  - El protocolo es **XML-RPC**. El método `procesarRPC` de la clase `Server` actúa como *dispatcher* central.
  - Publica métodos como: `login`, `getEstado`, `move`, `motors`, `gripper`, `emergencyStop`, y `resetEmergency`.
  - Clase `AdministradorSistema`: Un módulo específico para gestionar si el control remoto está habilitado o deshabilitado, persistiendo este estado en un archivo (`db/remote_state.dat`).
  - `main.cpp` maneja la subida de archivos (`/upload`): guarda el CSV, lo convierte a `.gcode` en `jobs/`, pero **no** lo ejecuta automáticamente. Para la ejecución, existe el método RPC `runFile`.
- **CLI administrativa**
  - `main.cpp` lanza un *thread* (`replThread`) para una **consola REPL** (**Read-Eval-Print Loop**).
  - Esta CLI **no** solicita login, pues es una consola de depuración local.
  - Permite ejecutar comandos directos (definidos en `buildCommandTable`) como `pkill` (para apagar el servidor), `motors on/off`, `exportLog` o enviar llamadas RPC manualmente (`rpc <metodo> [json]`).

Desde el punto de vista de POO, el servidor muestra el uso de:

- **Abstracción** del dominio (clases `EstadoRobot`, `RobotControllerSimple`, `Login`).

- **Composición** (El `RobotControllerSimple` *usa* una `ComunicacionControladorSimple`; `main` *compone* todos los servicios).
- **Gestión de recursos (RAII)** (Los destructores de `Login` y `ComunicacionControladorSimple` cierran la conexión a la BD y el puerto serie, respectivamente).
- Uso de `std::function` (en `buildCommandTable`) para un manejo flexible de comandos en la CLI.

2.2 Cliente con JS + HTML + CSS

Del lado cliente se desarrolló una **aplicación web** que corre en el navegador y se comunica con el servidor C++ mediante el mecanismo RPC expuesto.

La estructura general del cliente se organiza en:

- **HTML**
  Define la estructura de la interfaz:
  - Formularios de login.
  - Panel de control principal del robot (conexión, homing, movimientos, modos).
  - Sección para carga de archivos G-Code.
  - Consola de mensajes / log visible para el operador.
  - Elementos para indicadores de estado (textos, íconos, colores, etc.).
- **CSS**
  Se encarga del estilo:
  - Organización en paneles (barra lateral para comandos, panel central para estado).
  - Colores diferenciados para estados: conectado/desconectado, motores ON/OFF, modo manual/automático.
  - Estilos para realzar mensajes de error y advertencias.
- **JavaScript**
  Implementa la lógica del cliente. A partir del código existente se organizó el comportamiento en una clase principal (la clase `robotPanel.js` y la clase `server_watch.js`) que encapsula:
  - `robotPanel.js`: El estado actual del cliente (usuario autenticado, último estado reportado por el servidor, tarea en ejecución). Los métodos para invocar los servicios remotos:
    * `login()`, `logout()`
    * `connectRobot()`, `disconnectRobot()`
    * `sendHome()`, `sendMoveTo(x, y, z, v)`, `sendGripperOn/Off()`
    * `uploadGCode(file)`, `runTask(name)`
    * `refreshStatus()`, `fetchUserReport()`
  - `server_watch.js`: La actualización de la interfaz a partir de las respuestas del servidor:
    * Actualización de textos y clases CSS.
    * Habilitación / deshabilitación de botones según el estado (ej. no permitir nuevos movimientos durante una tarea automática).
    * Visualización de mensajes INFO/ERROR en una consola integrada.

Para mantener compatibilidad con versiones anteriores del frontend, se incluyó una pequeña **capa de compatibilidad** en los scripts, de forma que el nuevo objeto orientado a objetos

(`robotPanel`) pueda convivir un tiempo con las funciones globales existentes hasta migrar completamente al diseño modular.

Como implementación adicional, el cliente incorpora **efectos de sonido** asociados a eventos relevantes (inicio/parada de movimiento, activación de parada de emergencia, etc.).

2.3 Interfaz Gráfica

Aunque el servidor está preparado para funcionar sólo con interfaz tipo CLI, la solución incorpora una **interfaz gráfica de usuario web**, accesible desde cualquier navegador moderno.

La interfaz se estructura conceptualmente en tres zonas:

1. **Zona de conexión y autenticación**
   - Campos de usuario y contraseña.
   - Botón de login.
   - Indicador de rol (administrador / operador) y estado de acceso remoto.
2. **Panel de control del robot**
   - Botones para:
     - Conectar / desconectar el robot.
     - Hacer homing (`G28`).
     - Activar / desactivar motores.
     - Activar / desactivar el efector final (`M3` / `M5` o comandos equivalentes).
   - Campos de entrada para movimientos manuales:
     - Posición objetivo (`X, Y, Z`) y velocidad.
     - Selección de modo absoluto / relativo.
   - Área de estado:
     - Modo actual (manual / automático, absoluto / relativo).
     - Posición reportada (`M114`).
     - Último tiempo de movimiento informado.
3. **Gestión de tareas y reportes**
   - Sección para subir archivos de tareas en formato G-Code (`*.gcode`, `*.txt`, `*.csv` según implementación).
   - Lista de archivos disponibles para ejecución por usuario.
   - Botón para ejecutar la tarea seleccionada (modo automático).
   - Sección de **reportes**:
     - Vista del historial de órdenes de la sesión del usuario.
     - Vista de reportes administrativos con filtros básicos (p.ej., por usuario y rango de fechas).

Para el informe final se incluirán capturas de pantalla del navegador en funcionamiento, donde se evidencie:

- Login exitoso.
- Activación del robot y ejecución de homing.
- Ejecución de un movimiento manual.
- Carga y ejecución de un archivo de tareas.
- Visualización de reportes / logs.

2.4 Detalles de Instalación

En esta sección se describen los pasos necesarios para **compilar, configurar y ejecutar** el sistema en un entorno Linux (por ejemplo, Ubuntu), adaptando las rutas a lo que efectivamente se tenga en el repositorio.

Requerimientos de software

- **Servidor C++**
  - Compilador compatible con C++17 o superior (por ejemplo, `g++`).
  - Herramientas de build (`cmake` y/o `make`).
  - Biblioteca de XML-RPC.
  - Biblioteca de acceso a base de datos (SQLite).
  - Acceso al puerto serie donde se conecta el simulador del robot (Arduino configurado con el firmware provisto).
- **Cliente Web**
  - Navegador moderno compatible con ES6.
  - Servidor HTTP simple para servir los archivos estáticos de `/Code/client` (por ejemplo, `python3 -m http.server` o un pequeño servidor Node.js, según lo implementado).

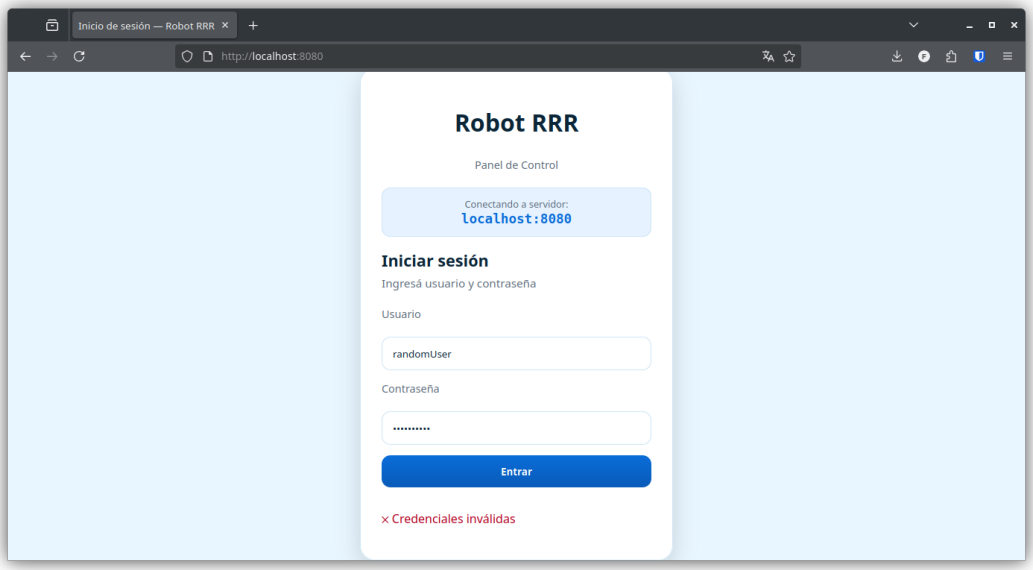
Estructura de directorios

A modo ilustrativo, la estructura de los códigos se organiza en una carpeta raíz `Code/`:

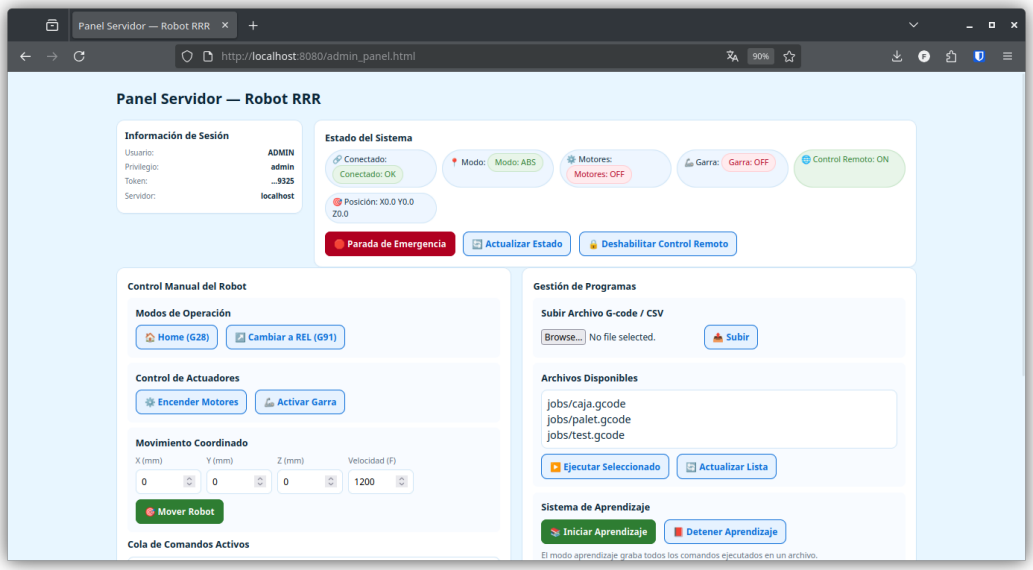
- `Code/server/`
  Contiene el proyecto en C++:
  - Código fuente (`src/`).
  - Headers (`include/`).
  - Archivos de configuración (por ejemplo, `config.json`, `server.cfg`).
  - Scripts de build (`CMakeLists.txt`, `Makefile`).
- `Code/HTML/`
  Contiene la aplicación web:
  - `index.html`, archivos HTML auxiliares.
  - Hojas de estilo (`css/`).
  - Scripts JavaScript (`js/`, incluyendo el módulo principal de la interfaz).
  - Recursos de audio e imágenes.
- `Code/db/`
  Scripts de creación de tabla de usuarios y otros datos auxiliares (en caso de usarse).

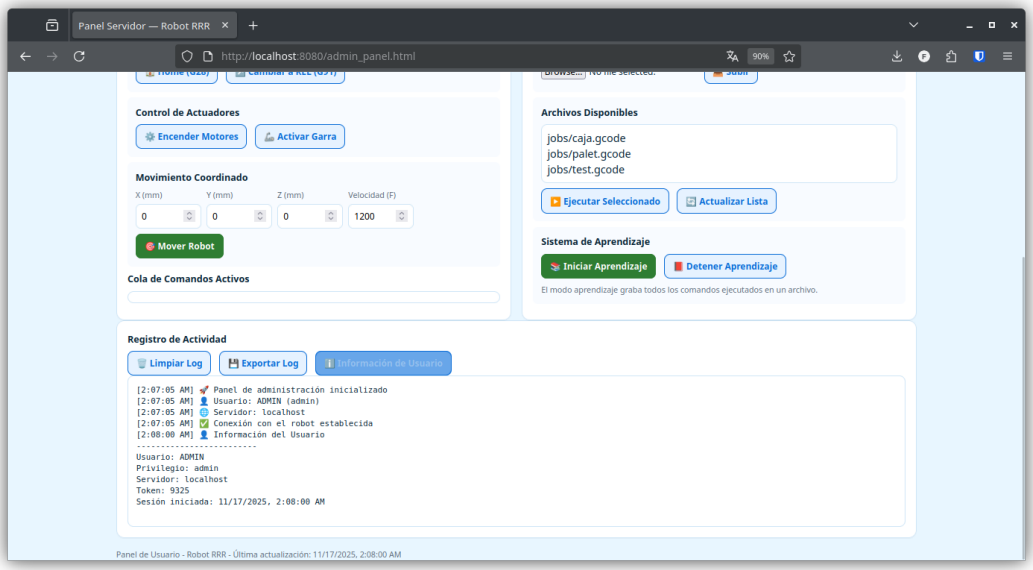
2.5 Imágenes de pruebas

Se incorporarán capturas de pantalla que documenten el comportamiento del sistema en escenarios representativos. A modo de guía, se proponen las siguientes figuras:

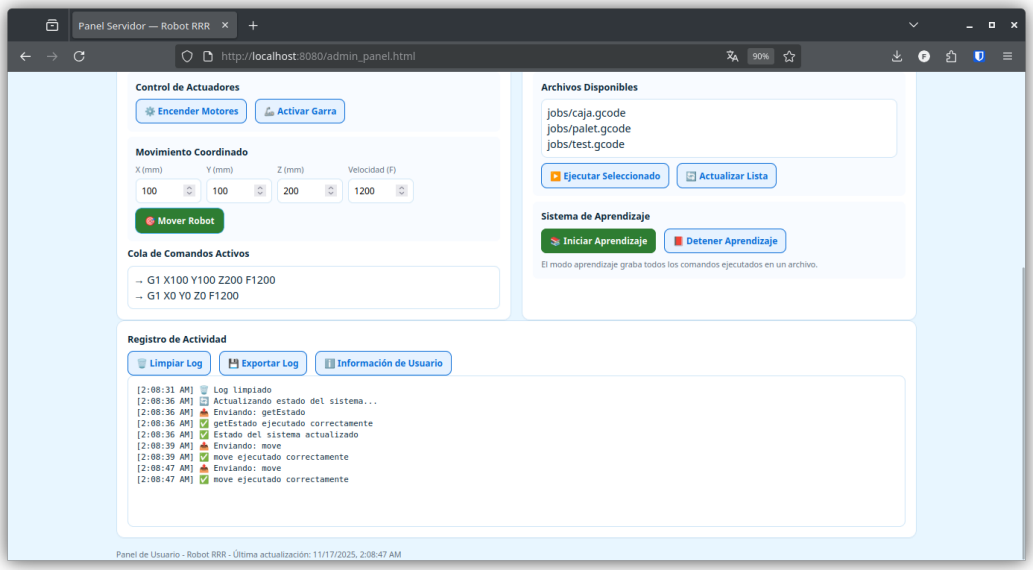


- **Figura 1.** Pantalla de login del cliente web, con validación correcta y mensaje de bienvenida.

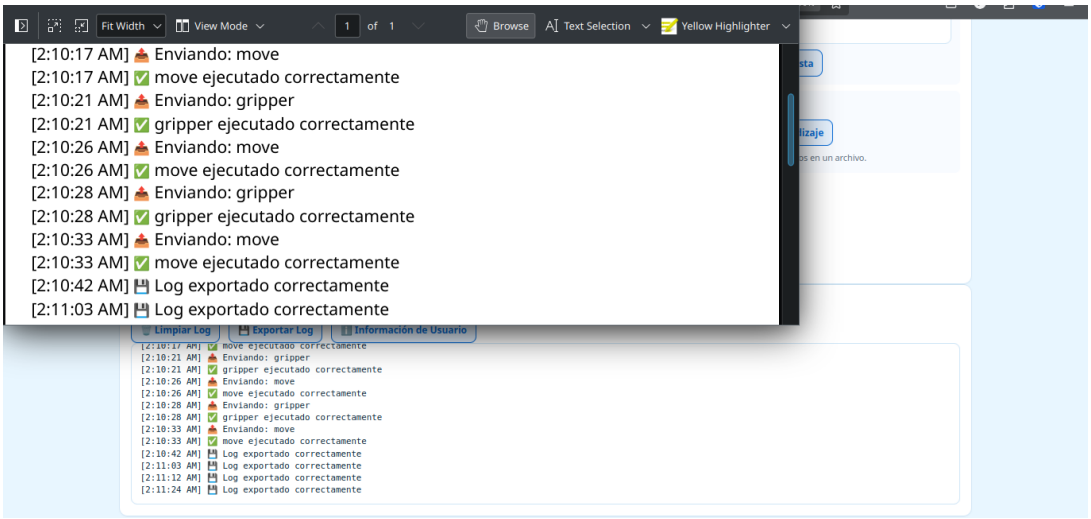




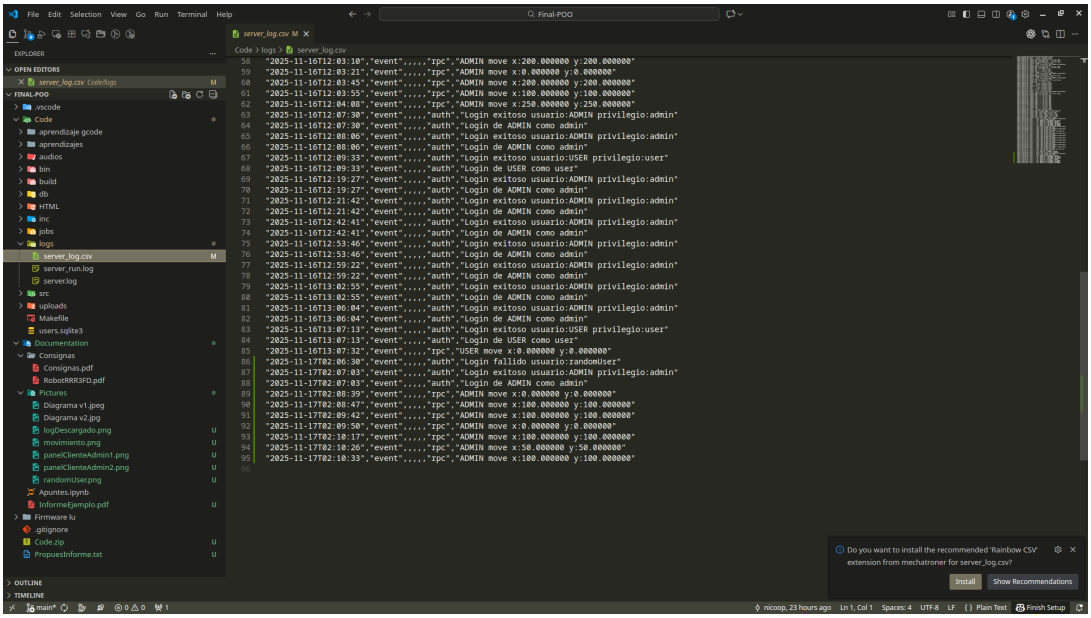
- **Figura 2.** Panel principal con el robot conectado, motores habilitados y posición indicada por el simulador.



- **Figura 3.** Ejecución de un movimiento manual, mostrando:



- **Figura 4.** Vista de un reporte de usuario (historial de comandos desde el último login).



- **Figura 5.** Vista de un reporte administrativo con filtros aplicados.

En todos los casos, las figuras se acompañarán con un breve pie explicativo que resuma qué se está probando, qué módulos intervienen y qué requisito de la consigna se está verificando.

3 Comentarios y Conclusiones

Debido a la condición de utilizar lenguaje C++ en el servidor y un lenguaje diferente en el cliente, se dio un buen entorno para experimentar con un poyecto multilenguaje. Debido a la simplicidad del resultado, la integración de código en C++ y código en JS no proporcionó ninguna dificultad adicional (más allá de entender las bases de JS para el paradigma de la Programación Orientada a Objetos).

Fue interesante buscar un punto intermedio entre la complejidad del diseño de la aplicación web y la cómodidad visual para el usuario final: se utilizaron “boxes” o “containers” de HTML

relativamente sencillos, pero se hizo un gran aprovechamiento de emojis para presentar una interfaz más amigable.

Debido al límite de tiempo, la utilización de pruebas unitarias presentó la mayor dificultad, por lo que la etapa de depuración de código fue bastante arcaica (prueba y error) en vez de utilizarse pruebas unitarias o integradas.

Otra dificultad importante estuvo relacionada con la **concurrencia y la robustez** del servidor: manejo de múltiples clientes, validación de credenciales, bloqueo de ciertas operaciones para usuarios no administradores, manejo de tiempos de espera del puerto serie y registro consistente en el log CSV. Esto llevó a incorporar manejo explícito de excepciones, códigos de estado y tests básicos de estrés.

Desde el punto de vista académico, el trabajo permitió aplicar de forma integrada: * Pilares de POO (abstracción, encapsulamiento, herencia, polimorfismo). * Diseño por capas y separación de responsabilidades. * Persistencia de datos (base de datos de usuarios, archivo CSV de log). * Comunicación en red mediante un mecanismo RPC. * Integración con un frontend web moderno.

Como líneas de trabajo futuro se identifican:

- Mejorar el sistema de **tests automatizados**, incluyendo pruebas unitarias de módulos C++ y tests de interfaz end-to-end.
- Extender el sistema de permisos con más roles y métricas de uso.
- Implementar notificaciones en tiempo real (por ejemplo, mediante WebSockets) para evitar que el cliente tenga que consultar periódicamente el estado del robot.

En conclusión, se logró una solución funcional que cumple el objetivo principal de **manipular de forma remota un robot 3DF con efector final**, respetando las consignas dadas e integrando la gran mayoría (si es que no todos) los módulos complementarios, aprovechando las capacidades de C++ y JavaScript en un contexto orientado a objetos.