Technical Report

Securing Multi-User Motor Control System

Paul Merlin
p.merlin.pro@hotmail.com

February 19, 2026

Contents

1 Introduction

The Sigmoïd motor system is currently controlled through a Python driver accessed by multiple users via JupyterLab. Because no access control or synchronization mechanism exists, several users may send commands simultaneously, leading to unsafe motor behavior, potential damage, and safety risks.

This document proposes a secure architecture ensuring exclusive, authenticated, and safe access to the motor.

2 Problem Statement

2.1 Current Issues

The following issues exist in the current architecture :

- Multiple notebooks can access the driver
- No authentication
- No concurrency control
- No safety validation
- No traceability

2.2 Risks

Therefore, the following risks can occur :

- Conflicting motor commands
- Mechanical damage
- User injury
- Loss of traceability

2.3 Definition of Done (DoD)

In that context, the Definition of Done (DoD) for this project is achieved when the multi-user motor control system is fully secured and prevents concurrent access under all normal operating conditions. The system must ensure that only authenticated and authorized users can send control commands through the API server. A distributed mutex mechanism must reliably guarantee exclusive access to the motor, including proper handling of timeouts, heartbeat renewal, and unexpected client disconnections. All motor commands must pass through the centralized backend, with no possibility of direct hardware access from user notebooks. The solution must be documented, tested, and deployed, with automated tests validating lock acquisition, release, failure recovery, and concurrency scenarios. In addition, the source code, deployment scripts, and documentation must be published in a public GitHub/GitLab repository, along with installation instructions and example usage. The project is considered complete only when the system operates safely in multi-user conditions, meets performance requirements, and has been successfully reviewed and validated by stakeholders.

3 Proposed Architecture

3.1 Overview

3.1.1 Concurrency management

Since the main challenge of the system is related to concurrency management, mutex mechanisms are introduced to guarantee that, at any given time, only one user is allowed to control the motor.

In the current architecture, different users access the motor from separate computers and through independent execution environments. Consequently, local mutexes, which are typically used to synchronize threads within a single process or on a single machine, are not sufficient to ensure mutual exclusion in this distributed context.

To address this limitation, a global mutex mechanism is required. This mutex is hosted on a centralized server and can be acquired by each user in turn. This approach ensures consistent synchronization across all clients, regardless of their physical location or execution environment.

Redis is used as the backend system for managing this global mutex. Redis is an in-memory key–value database, well suited for distributed synchronization. Each lock is typically implemented using atomic operations such as SETNX to acquire the lock and EXPIRE to set a time-to-live. This ensures that lock acquisition and release are safe, even across multiple clients, and prevents deadlocks in case a client crashes. It is well suited for distributed synchronization because it provides the following advantages:

- Centralized architecture, ensuring a single source of truth for lock ownership
- Network accessibility, allowing all clients and services to access the mutex remotely
- High performance and low latency, which minimizes the overhead introduced by synchronization mechanisms
- Support for atomic operations, guaranteeing that lock acquisition and release are performed safely without race conditions
- Built-in expiration mechanisms, enabling automatic lock release in case of client failure and preventing deadlocks

This mutex, stored with Redis, is managed exclusively by the central control server. Only this server is authorized to communicate with the motor driver, while users interact solely through a secured API interface.

This design ensures mutual exclusion, centralized validation, protection of the hardware interface, and full traceability of user actions. By preventing direct access to the driver, the system reduces the risk of unauthorized commands and accidental misuse.

3.1.2 Lock free mechanism

In the proposed architecture, Redis is responsible only for storing the distributed lock and enforcing automatic expiration through a time-to-live mechanism, ensuring that no lock can remain indefinitely in case of system failure. The control server manages user sessions and monitors activity by treating each motor command as a heartbeat signal. Whenever a user sends a valid command, the server verifies ownership of the lock, executes the request, updates the user's last activity timestamp, and renews the lock's expiration time in Redis. If no commands are received within a defined period, the server considers the session inactive, invalidates it, and explicitly releases the lock. In case of server failure while Redis is still running, the lock will eventually expire automatically, preventing deadlocks. Conversely, if Redis fails, the server must handle lock recovery or refuse new commands until Redis is available, ensuring safe operation. This mechanism allows sessions to remain active as long as the user is controlling the motor, while automatically revoking access in case of disconnection, application crashes, or network failures. Redis expiration acts as a secondary safety layer, guaranteeing lock release even if the server becomes unavailable, thereby preventing deadlocks and ensuring continuous system availability.

3.1.3 Authentication and Authorization

To ensure secure access to the motor control system, all users are required to authenticate through the Control API Server before being granted any operational privileges. Authentication is performed using secure credentials such as a username and password or short-lived API tokens. User credentials are stored as salted cryptographic hashes and are never transmitted or stored in plain text. All API communication is encrypted using HTTPS to prevent eavesdropping and replay attacks. Upon successful authentication, the server issues a secure session token that uniquely identifies the user for the duration of the session.

Authorization is enforced by associating each authenticated user with a predefined role and access level stored in the user database. These roles determine the operations that a user is permitted to perform, such as requesting control of the motor, executing movement commands, or accessing system logs. Before processing any command, the server verifies both the user's authentication status and authorization level. This layered security approach prevents unauthorized access, limits the impact of compromised accounts, and ensures that only approved users can interact with the motor control infrastructure.

3.1.4 Logging

Each authorized user is assigned a permanent unique identifier during the registration and authentication process, which remains consistent across sessions. A dedicated user database maintains the mapping between user IDs and identity information such as name, email, and access privileges. All motor control commands are recorded in a centralized logging database (PostgreSQL for instance), referencing the corresponding user ID, timestamp, command parameters, and execution status. This design ensures full traceability, accountability, and auditability, enabling administrators to monitor system usage, investigate incidents, and enforce access control policies.

3.1.5 Final architecture scheme

As illustrated in the system architecture, only the control server, after verifying the Redis lock, is allowed to communicate directly with the motor driver responsible for hardware interaction. Since all control commands are routed through the server, it acts as a centralized control point. This design enables the server to:

- Validate commands
- Enforce safety limits
- Check user permissions
- Monitor motor state
- Reject dangerous inputs

3.2 Server Architecture

In summary, the server works using these different tools :

- Framework: FastAPI

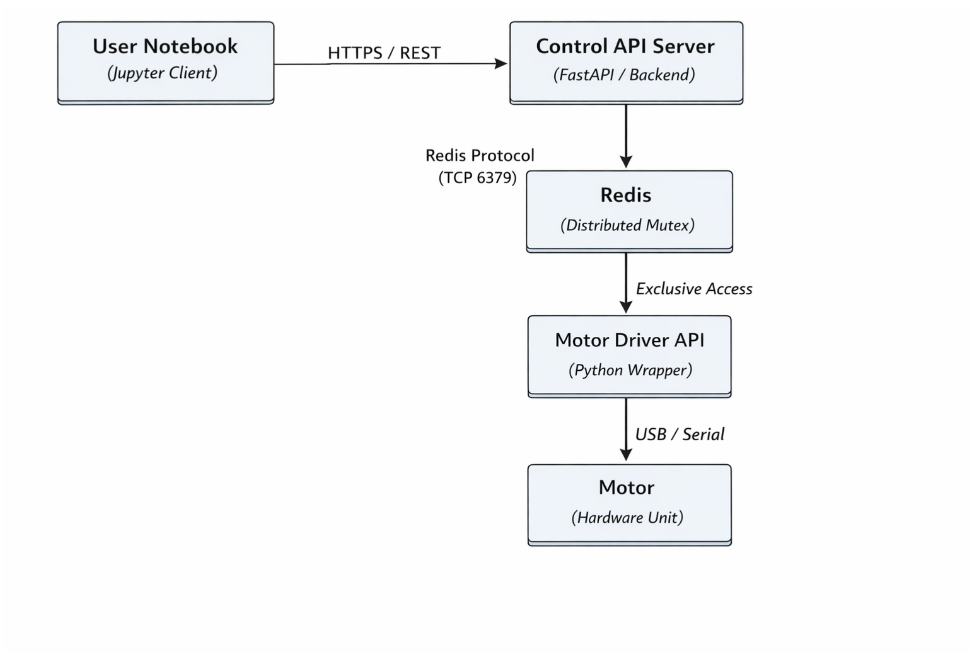

Figure 1: Secure Motor Control System Architecture

- Lock backend: Redis
- Containerization: Docker

3.3 Components

Below is a table to summarize every component's role of the new architecture :

4 Security Mechanisms

This section summarizes the security measures that have been set up.

4.1 Authentication

Each user authenticates using secure credentials such as a username and password or API tokens. Only authenticated users may access the API.

4.2 Exclusive Lock

A distributed lock ensures only one active controller.

- Lock owner
- Expiration time (lease)
- Heartbeat mechanism to keep the session alive while the motor receives valid commands
- Automatic release

Component	**Communication / Protocol**	**Role / Description**
User Notebook	HTTPS / REST	The user interacts via a Jupyter client, sending motor control commands and receiving feedback.
Control API Server	HTTPS / REST	Backend server (FastAPI) that exposes endpoints to control the motor, validates commands, orchestrates access to the motor hardware, and ensures safe operation.
Redis (Distributed Mutex)	Redis Protocol (TCP 6379)	Provides distributed locking to ensure exclusive access to the motor driver, preventing conflicting commands.
Motor Driver API	Exclusive Access / Python Wrapper	Python interface that communicates with the motor hardware, translating API commands into low-level instructions for execution.
Motor	USB / Serial	The physical hardware unit (motor) that executes movement commands.

Table 1: Components and roles in the secured motor control system

4.3 Command Validation

All commands are validated against:

- Maximum speed
- Maximum angle
- Acceleration limits
- Mechanical constraints

Unsafe commands are rejected.

4.4 Audit Logging

All commands are logged with:

- User identifier
- Timestamp
- Command
- Execution status

4.5 API Endpoints

The different endpoints exposed by the server are the following :

Method	Path	Description / Behavior
POST	/auth	Authenticate the user. Each subsequent request (lock, move, unlock) should be validated against the authenticated session.
POST	/lock	Request exclusive control of the motor. If the motor is already locked by another client, the server should return an error (e.g., HTTP 423 Locked or 409 Conflict) with information about the current lock holder and possibly a suggested retry time.
POST	/move	Send a motor command. Only allowed if the client currently holds the lock. The server should validate the command to avoid unsafe operations.
POST	/unlock	Release the lock on the motor, allowing other clients to acquire control.

5 Deployment

5.1 Containerization

The system uses containerization to isolate components and control hardware access.

In particular, the **Motor Driver** container is responsible for **communicating directly with the motor hardware** over USB/Serial. Its main role is to **execute motor commands**, but it **never decides or validates commands itself**. All validation, authentication, lock checks, and safety enforcement are performed by the **Control API Server** before any command reaches the motor driver.

- Only the **Motor Driver** container has access to hardware devices such as `/dev/ttyUSB0`. No other container or service can access the motor directly.
- The **Control API Server** container handles all incoming user commands via HTTPS/REST, verifies the distributed lock in Redis, validates command safety, and only then forwards commands to the Motor Driver API inside the Motor Driver Container.
- Containers that do not need hardware access run without USB privileges, following the principle of least privilege.
- This separation enforces hardware isolation, ensures traceability, and prevents accidental or malicious commands from being sent directly to the motor.

5.2 Docker Compose

Deployment orchestration is handled using **Docker Compose**, which manages multiple services and their dependencies in isolated containers:

- **Control API Server Container:** Runs the backend server exposing the API to control the motor. It handles authentication, validation, and lock checks. It requires network access to Redis but does not directly access the motor hardware.
- **Motor Driver Container:** Executes validated motor commands received from the Control API Server. It has exclusive USB access to the motor device.

- **Redis:** Provides a distributed locking mechanism to ensure exclusive access to the motor. All lock requests and coordination between clients pass through Redis.

Docker Compose allows services to run in isolated environments while still communicating over the network. This simplifies deployment, scaling, and reproducibility, and ensures that hardware access and critical operations are strictly controlled.

5.3 Security Considerations

- Only the motor API container has hardware privileges; other containers, such as Redis or user-facing services, cannot directly access the motor.
- Network communication between containers is limited to what is required (API calls and Redis connections), reducing the attack surface.
- The architecture enforces **exclusive control and authentication mechanisms** consistently across all containerized services.

6 Testing Strategy

The key tests to be performed are as follows :

6.1 Unit Tests

- Lock acquisition
- Lock release
- Authentication
- Validation rules
- Command syntax and range checking
- Error handling for invalid requests
- Logging and audit trail verification

6.2 Integration Tests

- Multiple concurrent users
- Crash recovery and restart behavior
- Timeout handling for locks
- Interaction between API server and Redis lock
- Correct command execution through Motor Driver API
- Session management and token expiration

6.3 Stress Tests

- Parallel access attempts to the motor
- Network failure simulation
- High-frequency command bursts
- Redis failure / unavailability simulation
- Hardware disconnection and reconnection handling
- API response under heavy load

6.4 Security Tests

- Unauthorized access attempts
- Privilege escalation attempts
- Injection attacks (e.g., malicious commands)
- Replay attacks for lock acquisition

6.5 Regression Tests

- Ensure new code changes do not break existing functionality
- Verify API backward compatibility
- Confirm previously fixed bugs do not reappear

7 AI Usage Declaration

Artificial intelligence tools were used to validate my choices related to :

- General architecture
- Pseudocode

I used AI to validate solutions, allowing it to propose answers rather than requesting them directly.

7.1 Example Prompts

> "Is Redis a suitable database for managing a lock between multiple users on separate computers?"
>
> "Please review this pseudo code for handling locks (acquire, release, etc.). Are there any potential issues or improvements?"

All generated content was reviewed and adapted manually.

8 Conclusion

This solution introduces a secure, scalable, and safe architecture for controlling shared hardware resources. By centralizing access, enforcing authentication, and implementing safety mechanisms, the system significantly reduces operational risks.