Technical Report

Securing Multi-User Motor Control System

Paul Merlin
p.merlin.pro@hotmail.com

19 février 2026

Table des matières

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 main following issues are present 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 Mutex

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 that 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

TODO : explain lock free mechanism in a short paragraph, as we said before : we do not only rely on time, the lock is kept while the user sends commands to the motor through the server. Only when the user having the lock become "inactive", the lock is freed

3.1.3 User authentication

User authentication is implemented through a token-based mechanism, such as JSON Web Tokens (JWT). After successful authentication, users receive a signed token that must be included in all subsequent requests. The server verifies this token before granting access to the motor control functionalities and applies role-based authorization policies.

This combination of centralized access control, distributed locking, and strong authentication guarantees safe, reliable, and accountable operation of the shared motor system.

3.1.4 Logging

EXPLAIN SERVER LOGGING COMMANDS

As a centralized Motor Control Server (MCS) is introduced to manage all motor access, here is a diagram of the new architecture :

TODO : INCLUDE PICTURE

Only the server communicates directly with the hardware. Since all commands pass through the server, it becomes a control point. It can :

— Validate commands

— Enforce safety limits
— Check user permissions
— Monitor motor state
— Reject dangerous inputs

3.2 Components

TODO : fix the table, according to the image of the new architecture.

Component	Role
Jupyter Client	Sends commands
API Server	Authentication and routing
Lock Manager	Exclusive access
Safety Layer	Command validation
Driver Wrapper	Hardware control
Logger	Traceability

4 Security Mechanisms

4.1 Authentication

Each user authenticates using JWT 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)
— Automatic release

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

5 Software Design

5.1 Server Architecture

— Framework : FastAPI

— Lock backend : Redis

— Containerization : Docker

5.2 Core Classes

5.2.1 MotorManager

```
class MotorManager:

    def acquire_lock(self, user):
        pass

    def release_lock(self, user):
        pass

    def execute(self, user, command):
        pass
```

5.3 API Endpoints

Method	Path	Description
POST	/auth	Login
POST	/lock	Acquire control
POST	/move	Send command
POST	/unlock	Release control

6 Deployment

6.1 Containerization

Only the motor container has access to the USB interface.

```
devices:
  - /dev/ttyUSB0
```

This prevents unauthorized hardware access.

6.2 Docker Compose

```
services:
  motor-api:
    build: .
    privileged: true

  redis:
    image: redis
```

7 Testing Strategy

7.1 Unit Tests

— Lock acquisition
— Authentication
— Validation rules

7.2 Integration Tests

— Multiple concurrent users
— Crash recovery
— Timeout handling

7.3 Stress Tests

— Parallel access attempts
— Network failure simulation

8 Definition of Done

8.1 Functional

— Authentication implemented
— Exclusive access guaranteed
— Automatic lock expiration
— Safe command execution

8.2 Safety

— Motion limits enforced
— Emergency stop available
— Watchdog monitoring

8.3 Quality

— Dockerized deployment
— Documentation completed
— Automated tests
— Continuous integration

9 AI Usage Declaration

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

— General architecture
— Pseudocode

I have always used AI for validation purposes (therefore, proposing a solution instead of asking for it).

9.1 Example Prompts

"Is Redis proper database to manage a lock between several users operating from separate computers ?"

"Validate this pseudo code, which aims at handling the lock (acquire, release, ...)"

All generated content was reviewed and adapted manually.

10 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.

TODO : "explain that" : Future improvements may include hardware-level safety systems and predictive maintenance features.