UDPServerManager – Project Plan & Architecture (Updated)

Project Overview

UDPServerManager **is** the Supervisor function. It is a Windows 10/11 desktop application built with PySide6 (Qt for Python) that provides a modern GUI for monitoring, configuring, and interacting with a fleet of servers (Pico/xiRTOS modules, Raspberry Pi 4B, and AS-410M) over UDP and TCP. The Supervisor (UDPServerManager) is strictly administrative and does not make control decisions.

The Supervisor (UDPServerManager) exposes both TCP and UDP interfaces to a higher-level entity: the **SpoolerController**. The SpoolerController is a separate program that interfaces with the Human Machine Interface (HMI) and performs all command and control functions for the entire MZSpooler system. The SpoolerController issues commands and receives status/data via the Supervisor, which acts as the administrative and communication hub for the lower-level servers.

Most devices will be on the same local network (LAN, wired or wireless). However, the system must also support at least one device, the AS-410M, located on a separate network, accessible only via firewall or VPN traversal. The AS-410M will function as a UDP-only server and will report through the Supervisor (UDPServerManager), ensuring unified management, logging, and protocol handling. Special handling for secure communication, NAT traversal, or VPN configuration may be required for the AS-410M.

Project Goals

- Deliver a robust, intuitive GUI for UDP-based device management.
- Support real-time monitoring, configuration, and diagnostics.
- Enable easy extension for future protocols (e.g., TCP, AI/ML integration).
- Ensure maintainability, testability, and modern UX.
- Provide a unified interface for both Pico and Pi 4B servers.
- Provide an easy, reliable interface to the AS-410M device, including secure network traversal and unified management.

Key Features & User Stories

Core Features

- Device Discovery: Scan and list available Pico/xiRTOS and Pi 4B servers on the network.
- Connection Management: Connect/disconnect to multiple devices; show connection status.
- Live Data Monitoring: Display real-time telemetry, logs, and images from devices.
- Command/Config Interface: Send commands and configuration data to devices (shared dictionary/schema).
- Logging: Save and view communication logs.
- File/Image Transfer: Use TCP for large data (images, video, logs) from Pi 4B servers.
- Settings: Manage network settings, device profiles, and app preferences.

- Error Handling: User-friendly error messages and reconnection logic.

User Stories

- As a user, I can scan for all server types and see their status in a list.
- As a user, I can select a device to view its live data, logs, and images.
- As a user, I can view reduced frame rate images from one or more cameras.
- As a user, I can send commands or configuration updates to any device.
- As a user, I can view and export communication logs and images for troubleshooting.
- As a user, I can adjust network and application settings via a settings dialog.

Main Architectural Components

1. UI Layer (PySide6/Qt): Main window, device list, detail panels, settings dialogs, log/image viewers.
2. Networking Layer: Async UDP client (all servers), TCP client (for file/image transfer), device discovery, protocol handling.
3. Async Integration: Use asyncio for non-blocking networking, integrate with Qt event loop (qasync).
4. Configuration & Persistence: Store user settings, device profiles, and logs (JSON, INI, or SQLite).
5. Logging & Diagnostics: Centralized logging (to file and UI), error reporting.
6. Extensibility: Modular architecture for future protocol/device support, AI/ML integration.

Proposed Folder/File Structure

```
UDPServerManager/
│
├── main.py                 # Application entry point
├── requirements.txt        # Python dependencies
├── README.md
│
├── app/
│   ├── ui/                 # UI files (Qt Designer .ui, QML, resources)
│   ├── widgets/            # Custom PySide6 widgets
│   ├── views/              # Main window, dialogs, panels, image/AI viewers
│   └── resources/          # Icons, images, etc.
│
├── core/
│   ├── networking.py       # UDP/TCP client, discovery, protocol logic
│   ├── file_transfer.py    # TCP file/image transfer logic
│   ├── async_integration.py # Qt/asyncio event loop bridge
│   ├── config.py           # Settings, profiles, persistence
│   ├── logger.py           # Logging utilities
│   ├── device.py           # Unified device model (capabilities, type)
│   └── models.py           # Data models (Device, Message, etc.)
│
├── tests/
│   ├── test_networking.py
│   ├── test_config.py
│   └── ...
```

```
│
└── docs/
    └── project_plan.md      # This document
```

Best Practices for PySide6 & asyncio Integration

- Use qasync to bridge Qt and asyncio event loops for smooth, non-blocking UI/networking.
- Keep UI logic and networking logic separated (MVC/MVVM pattern recommended).
- Use signals/slots for UI updates from async tasks.
- Avoid blocking calls in the UI thread.
- Use type hints and docstrings for maintainability.
- Modularize code for testability and future extension.
- Use a unified command/config schema for all server types.

Recommended Libraries & Patterns

- qasync: For Qt/asyncio integration.
- pydantic/dataclasses: For data models and validation.
- pytest: For testing.
- loguru or Python logging: For robust logging.
- QSettings or configparser: For settings persistence.
- Qt Designer: For rapid UI prototyping.

Roadmap

MVP (Milestone 1)

- Basic UI: Device list, connect/disconnect, live log panel.
- Async UDP networking with device discovery.
- Command/config send/receive (shared schema).
- Settings dialog and basic persistence.
- Logging to file and UI.

Milestone 2

- TCP file/image transfer for Pi 4B servers.
- Image/video viewer panel in UI.
- Unified device abstraction (Pico, Pi 4B, future types).

Milestone 3

- AI/ML integration for image recognition (optional, future).
- Plugin system for new device types/protocols.
- User authentication and role management.
- Auto-update mechanism.
- Improved error handling and diagnostics.

- Cross-platform support (Linux/macOS).

Hierarchical System Context

The Supervisor (UDPServerManager) will operate as part of a hierarchical control system. In this role, it will also provide both TCP and UDP interfaces to a next-higher-level PC program, referred to as the SpoolerController. This upper-level SpoolerController application will:

- Run on a PC with a touch-screen Human Machine Interface (HMI)
- Feature a simple UI (On/Off button, Next/Back button, display of operating instructions or static images)
- Communicate with the Supervisor (UDPServerManager) for status, control, and data exchange
- Not require complex UI logic—focus is on reliability and clear operator feedback
- Receive data and status from the AS-410M device via the Supervisor (UDPServerManager)

> **Note:** Details of the SpoolerController application functions, UI, and protocols will be documented in a separate document for clarity and modularity.

Project Architecture: Supervisor, SpoolerController, and Edge Server Threads

Overview

This system manages multiple UDP and TCP edge servers using a scalable, multi-threaded architecture. The design supports high concurrency, low-latency control, and robust separation of concerns.

Main Components

1. Supervisor Thread (Main/UI Thread)

- **Administrative Only**: The Supervisor is strictly an administrative and monitoring function. It does **not** make any control decisions.
- **Role**: Owns and updates the Qt GUI (PySide6), periodically queries each edge server thread for status/heartbeat data, and displays all status and metrics to the user.
- **No Control Routing**: The Supervisor does **not** route control directives or emergency messages. It may relay messages or display status as needed, but never initiates control actions.
- **Naming**: While the main module is called `UDPServerManager`, it is functionally the **Supervisor** as described here.

2. SpoolerController (HMI)

- **Control Authority**: The SpoolerController is the sole decision-maker for all control logic in the system.
- **Direct Communication**: Sends control directives directly to edge server threads via their standard mailboxes.
- **Emergency Handling**: Places emergency/broadcast messages in a **shared emergency mailbox**.

- **Bypass for Time-Critical Data**: Can request time-critical data directly from edge server threads (bypassing Supervisor).
- **Flexible Routing**: May also communicate through the Supervisor if UI or administrative intervention is required.

3. Edge Server Threads (One per Managed Server)

- **Thread-per-Server**: Each runs as a lightweight Python thread (not QThread).
- **Local State**: Maintains its own local data (status, heartbeat, etc.).
- **Mailboxes**: Has a **standard mailbox** (queue) for per-server control messages. All threads share a **single emergency mailbox** (queue) for urgent/broadcast messages.
- **Message Handling**: Each thread checks the emergency mailbox first, then its own standard mailbox. If an emergency message is relevant, the thread acts on it immediately.
- **No Qt Access**: No direct access to the Qt GUI.

Mailbox Pattern

- **Standard mailbox:** `queue.Queue()` per edge server thread
- **Emergency mailbox:** One shared `queue.Queue()` for all edge server threads
- **Direct Control**: The SpoolerController can:
  - Send a control message to a specific server's standard mailbox
  - Place an emergency/broadcast message in the shared emergency mailbox (all edge servers check this)
- Edge server threads always check the emergency mailbox first, then their own

Data Flow

- Edge server threads update their own local data structures
- Supervisor periodically pulls status/heartbeat from each thread for display
- SpoolerController sends control/emergency messages directly to threads (never through Supervisor)

Benefits

- Highly scalable (dozens of threads on modern CPUs)
- Fast, direct control and emergency response
- Clean separation of UI, control, and server logic
- Thread-safe, robust, and easy to extend

This architecture can be further extended to support additional features, such as logging, diagnostics, or advanced scheduling, as needed.

Current List of PICO2W-Based Servers

Server Name	Description/Role	Notes
TransferTapeBrake	Controls tape brake mechanism	

Server Name	Description/Role	Notes
SpeedSensor (transfer)	Measures speed (transfer path)	One instance for transfer
SpeedSensor (transport)	Measures speed (transport path)	One instance for transport
capstanDrive	Controls main capstan drive	
frameControl	Controls frame positioning/mechanics	
pressurePlateControl	Controls pressure plate actuator	
transportSpoolDrive	Controls transport spool (capstanDrive-based)	Inherits motor control from capstanDrive
windowSpoolDrive	Controls window spool (capstanDrive-based)	Inherits motor control from capstanDrive
wasteSpoolDrive	Controls waste spool (capstanDrive-based)	Inherits motor control from capstanDrive
dancerController (input)	Controls input dancer mechanism	Identical code to waste dancerController
dancerController (waste)	Controls waste dancer mechanism	Identical code to input dancerController

- All servers are based on the PICO2W platform and communicate via UDP.
- Some drives (transport, window, waste) reuse the capstanDrive logic with specific configuration.
- There are two SpeedSensor instances: one for transfer, one for transport.

Current List of Raspberry Pi 4B-Based Servers

Server Name	Description/Role	Notes
QA Sensor	Quality assurance sensor (e.g., image/AI tasks)	Compute-intensive, TCP for images
SamplePositionSensor	Detects/monitors sample position	Compute-intensive, TCP for images
FramePositionSensor	Detects/monitors frame position	Compute-intensive, TCP for images
FrameQASensor	Frame quality assurance sensor	Compute-intensive, TCP for images

- All servers are based on the Raspberry Pi 4B platform.
- All are compute-intensive and use TCP for image transfer as needed.

Current List of External Servers

Server Name	Description/Role	Notes
AS-410M	DNS coordination and sample meta data	On wired LAN, Firewall/VPN

This plan provides a clear, actionable foundation for development and onboarding. Update as the project evolves.