UDP Server Manager Architecture

Document Version: 2.0
Last Updated: February 19, 2026
Target: UDP Server Manager v2.0+

Overview

This project is a modular Supervisor for managing UDP-connected devices. Each device type has its own worker, handler, and command dictionary, allowing for scalable and maintainable code. The application features a modern Qt-based GUI with advanced command management, real-time status monitoring, and multimedia capabilities.

System Architecture

High-Level Structure

```
UDP Server Manager
├── Application Layer (main.py)
├── UI Layer (app/ui/)
│   ├── Main Window (gui.py)
│   ├── Device Panel (device_panel.py)
│   ├── Message Creator Panel (message_creator_panel.py)
│   └── Status Panel (status_panel.py)
├── Core Layer (core/)
│   ├── UDP Networking (udp.py)
│   └── Workers (workers/)
│       └── Device-specific implementations
├── Shared Dictionaries (shared_dictionaries/)
│   └── Command Dictionaries (JSON)
└── Data (data/)
    └── Server Configurations (servers.json)
```

Key Components

Application Core

- **main.py**: Application entry point
  - Loads server configurations from `data/servers.json`
  - Initializes GUI with server list and status icons
  - Manages application lifecycle
- **config.py**: Centralized configuration
  - Window dimensions and layout parameters
  - UI component sizes (interactive frame: 350px, status frame: 350px, logging frame: 200px)

    - Styling constants for consistent UI appearance

Networking Layer

- **core/udp.py**: UDP networking engine
    - `UDPClientThread`: Worker thread for UDP communication
    - Single UDP socket per device for bidirectional communication
    - Automatic port binding and management
    - Thread-safe message queue and mailbox system
    - Signal-based communication with GUI

User Interface

The application features a three-tier vertical layout:

1. Interactive Frame (Top - 350px)

Layout: Three-column horizontal split

Left Column - Device Panel (`app/ui/device_panel.py`)

- Device selection dropdown by location
- Server list with status indicators
- Connection status visualization
- Width: 200px

Center Column - Message Creator Panel (`core/workers/capstanDrive/message_creator_panel.py`)

- **Hierarchical Command Menu System:**
    - Grouped flyout menus by command category:
        - Device Control (LED, HPL, STEPPER, ENCODER)
        - Device Status (GET_LED, GET_HPL, GET_STEPPER, GET_ENCODER, GET_ALL)
        - Error Handling (CLR_WARN, GET_ERROR_LOG, CLEAR_ERROR_LOG, etc.)
        - System Administration (SET_TIME, GET_TIME_STATUS)
    - Context-sensitive parameter display (up to 10 parameters)
    - Real-time message assembly
    - Dropdown-based enum selection
    - Type-validated input fields
    - Conditional parameter visibility based on command context

Right Column - Send/Reply Panel

- Send button for command transmission
- Request box showing sent commands
- Reply box displaying device responses
- Width: 200px

2. Status Frame (Middle - 350px) [NEW]

Component: `app/ui/status_panel.py`

The Status Panel provides two switchable display modes:

Table Mode:

- Two-column table (Item Name | Item Data)
- Sortable and searchable
- Alternating row colors for readability
- Dynamic content updates

Split Mode:

- Left: Text box for status messages and device information
- Right: Media display area supporting:
  - **Static Images:** PNG, JPG, BMP, GIF
  - **Video Playback:** MP4, AVI, MOV with full controls
    - Play/Pause/Stop buttons
    - Seekable timeline slider
    - Time display (current / total duration)
    - Aspect ratio preservation
  - **Network Streams:** HTTP/HTTPS/RTSP URLs (basic support)
  - **Future:** Live streaming from Raspberry Pi 4B devices

3. Logging Frame (Bottom - 200px)

- Read-only text area for application logs
- Auto-scroll to latest messages
- Timestamped entries
- UDP communication tracking

Worker/Handler Pattern

Each device type follows a modular pattern:

Structure:

```
core/workers/deviceType/
├── deviceType_worker.py          # Worker thread implementation
├── deviceType_handler.py         # Command execution logic
├── deviceType_config.py          # Device-specific constants
├── deviceType_commandDictionary.json  # Command definitions
└── message_creator_panel.py      # Device-specific UI (optional)
```

Components:

- **Worker Thread**: Manages UDP communication and command routing
- **Handler**: Implements device-specific command logic
- **Config**: Device constants (timeouts, defaults, instance mappings)
- **Command Dictionary**: JSON-based command definitions with:
  - Command categories for menu organization

- Parameter specifications (type, range, conditions, units)
- Instance configurations
- Validation rules

Command Dictionary System

Command dictionaries are JSON files that define device capabilities:

Structure:

```
{
  "categories": [
    "Device Control",
    "Device Status",
    "Error Handling",
    "System Administration"
  ],
  "commands": {
    "COMMAND_NAME": {
      "category": "Device Control",
      "description": "Command description",
      "param_count": 3,
      "parameters": [
        {
          "name": "param_name",
          "type": "enum|integer|float|boolean|string",
          "param_number": 1,
          "options": ["option1", "option2"],
          "condition": ["parent_param == value"],
          "range": {"min": 0, "max": 100},
          "units": "ms",
          "notes": "Additional information"
        }
      ],
      "function": "handler_function_name"
    }
  }
}
```

Features:

- Category-based command organization for hierarchical menus
- Conditional parameter display based on parent parameter values
- Type validation and range checking
- Unit specifications for clarity
- Enum-based options for controlled input

Communication Flow

```
User Interaction → UI (Message Creator)
                        ↓
                Command Assembly
                        ↓
           UDP Client Thread (Worker)
                        ↓
                Device Handler
                        ↓
           UDP Socket (Send/Receive)
                        ↓
                Network Device
                        ↓
           Response Processing
                        ↓
      GUI Update (Reply Box, Status Panel, Log)
```

Detailed Flow

1. **Device Selection:**
    - User selects device from Device Panel
    - GUI creates UDP worker thread for selected device
    - Local ephemeral port assigned for bidirectional communication
2. **Command Creation:**
    - User selects command from hierarchical menu
    - Message Creator displays relevant parameters
    - Real-time message assembly with validation
3. **Message Transmission:**
    - User clicks SEND button
    - Command queued in worker thread
    - UDP packet sent to device
    - Local port logged for response tracking
4. **Response Handling:**
    - Worker thread receives UDP response on same socket
    - Handler processes response
    - GUI updated via Qt signals:
        - Reply box shows device response
        - Status panel updated with device state
        - Log panel records transaction
5. **Status Display:**
    - Status Panel shows device information in table or split mode
    - Video/images displayed for visual feedback

- Real-time updates as device state changes

Error Handling

Network Errors

- **Port Mismatch Detection:** Log analysis tools for debugging UDP port issues
- **Timeout Handling:** Configurable timeouts per command type
- **Retry Logic:** Automatic retry for transient network failures
- **Connection State Tracking:** Visual indicators for connection health

Command Errors

- **Validation:** Pre-send validation of parameters
- **Malformed Commands:** Handler-level error detection
- **Device Errors:** Error message parsing and display
- **Log Context:** Detailed error logging with stack traces

UI Error Handling

- **Graceful Degradation:** UI remains responsive during errors
- **User Feedback:** Clear error messages in appropriate UI panels
- **Recovery Options:** Retry mechanisms and connection reset

Extending the System

Adding a New Device Type

1. **Create Worker Structure:**

```
core/workers/newDevice/
├── __init__.py
├── newDevice_worker.py
├── newDevice_handler.py
├── newDevice_config.py
├── newDevice_commandDictionary.json
└── message_creator_panel.py (if custom UI needed)
```

2. **Define Command Dictionary:**
   - Create JSON file with command definitions
   - Organize commands into logical categories
   - Specify parameters with types and validation rules
3. **Implement Handler:**
   - Subclass base handler if available
   - Implement command execution functions
   - Add error handling and validation

4. **Configure Server:**
    - Add entry to `data/servers.json`
    - Specify device type, IP, port, and location
5. **Test Integration:**
    - Start application and select new device
    - Verify command menu population
    - Test command execution and response handling

Customizing UI Components

Status Panel Customization:

```
# Add custom status display
main_window.update_status_table({
    'Device': 'NewDevice',
    'Status': 'Active',
    'Custom Field': 'Value'
})

# Display device image
main_window.update_status_image('path/to/device_image.png')

# Display instructional video
main_window.update_status_video('path/to/tutorial.mp4')
```

Message Creator Customization:

- Extend `MessageCreatorPanel` class for device-specific UI
- Override parameter rendering for custom controls
- Add device-specific validation logic

Performance Considerations

UI Responsiveness

- Worker threads prevent UI blocking during network operations
- Qt Signal/Slot mechanism for thread-safe GUI updates
- Lazy loading of command dictionaries
- Efficient message assembly algorithms

Network Optimization

- Single UDP socket per device reduces port consumption
- Configurable buffer sizes for different network conditions
- Ephemeral port binding for proper NAT traversal
- Minimal packet overhead in protocol design

Memory Management

- Bounded log buffer to prevent memory growth
- Cleanup of disconnected worker threads
- Efficient video decoding with Qt Multimedia hardware acceleration
- Resource release on application shutdown

Security Considerations

Current Implementation

- Local network operation assumed
- No authentication layer (devices trusted)
- Clear-text UDP communication

Future Enhancements (Raspberry Pi 4B Phase)

- Device authentication via shared secrets
- Encrypted UDP payloads (DTLS)
- Certificate-based device identification
- Access control lists for command authorization
- Secure video streaming (HTTPS/SRTP)

Future Architecture Evolution

Raspberry Pi 4B Integration

Planned Architecture:

```
PC Supervisor ↔ [Network] ↔ Pi 4B Edge Devices
     ↑                            ↑
     ↓                            ↓
Status Panel                 Pi Camera Module
Video Display                Video Streaming Server
                             Hardware H.264 Encoding
```

Key Additions:

- Real-time video streaming from Pi Camera modules
- WebRTC support for ultra-low latency (<100ms)
- Multi-camera management and layout controls
- Stream health monitoring and auto-reconnection
- Recording capabilities for audit trails

See `docs/live_streaming_roadmap.md` for detailed planning.

Scalability Enhancements

- Support for multiple simultaneous devices

- Device discovery via mDNS/Bonjour
- Command batching and macro support
- Scriptable automation interface
- REST API for third-party integration

Related Documentation

- **UI Components:** See `docs/ui_components_guide.md` for detailed UI documentation
- **Status Panel:** See `docs/status_panel_guide.md` for multimedia capabilities
- **Command Menus:** See `docs/command_menu_system.md` for menu customization
- **Usage Guide:** See `docs/usage.md` for end-user instructions
- **Contributing:** See `docs/contributing.md` for development guidelines
- **Live Streaming:** See `docs/live_streaming_roadmap.md` for future features

For PDF Generation: This document is formatted for professional PDF output using `docs/pdf-style.css`. See `docs/pdf_generation_guide.md` for instructions.