Real-Time Production Line Sensor Dashboard

Project Presentation

Si-Ware System Assessment Project

Project Overview

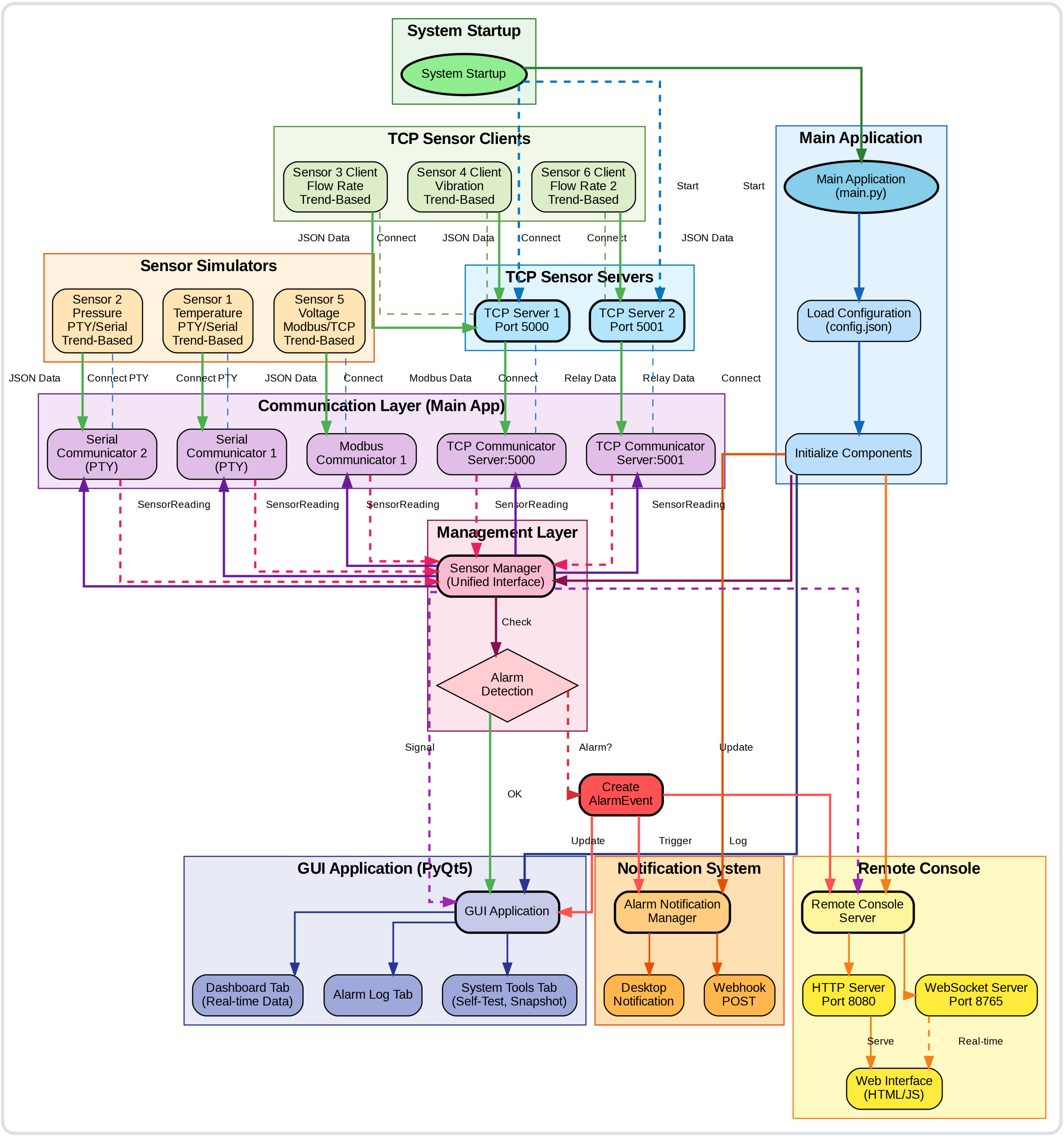
Real-Time Production Line Sensor Dashboard is a comprehensive real-time monitoring and maintenance system for industrial production lines.

This project was developed as part of the **Si-Ware System Assessment**, demonstrating advanced software engineering practices, multi-protocol communication, and cross-platform development capabilities.

Key Capabilities

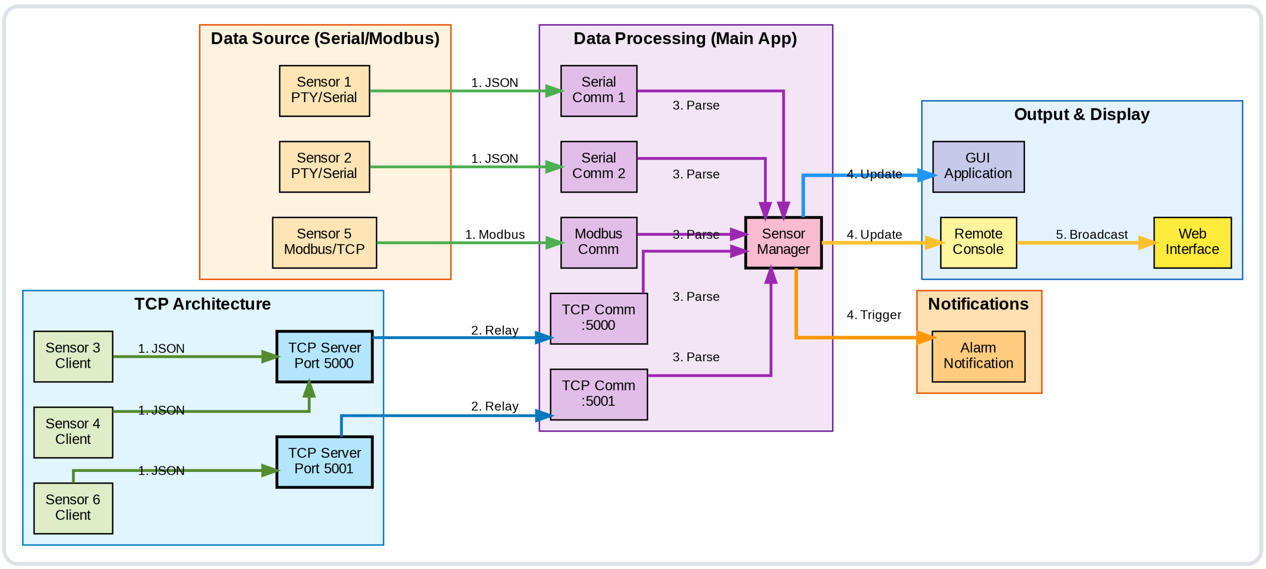
- Real-time monitoring of multiple sensors across different communication protocols
- Intelligent alarm management with automatic detection
- Desktop and web-based interfaces for monitoring and maintenance
- Cross-platform support (Linux and Windows)
- Remote access via WebSocket with authentication

System Architecture



Complete system architecture showing: - Sensor Simulators (Serial, TCP, Modbus) - Communication Layer (Multi-protocol support) - Management Layer (Sensor Manager, Alarm System) - GUI Application (PyQt5 Desktop Interface) - Remote Console (WebSocket Server) - Notification System (Webhook & Desktop)

Data Flow



Data flow sequence: 1. Sensor simulators generate data 2. Communication layer receives data via multiple protocols 3. Sensor Manager processes and validates data 4. Alarm system checks limits and triggers notifications 5. GUI updates in real-time 6. Remote console provides web access

Software Flow: Sensor to GUI/Web Interface

Thread Architecture

Worker Thread Model: - One worker thread per unique communication endpoint (not per sensor) - Serial: One thread per unique port (e.g., `/dev/pts/7`) - TCP: One thread per unique server (e.g., `localhost:5000`) - Modbus: One thread per unique server (e.g., `localhost:1502`)

Example: - 5 sensors on 3 ports = 3 worker threads - 4 TCP sensors on 2 servers = 2 worker threads - Mixed: 3 serial (2 ports) + 4 TCP (2 servers) + 2 Modbus (1 server) = 5 total threads

Data Flow from Sensor to GUI

Step 1: Sensor Data Reception (Worker Thread)

```
Sensor Simulator → Physical/Network Connection → Worker Thread
```

- Worker thread reads data from port/socket (blocking I/O)
- Data received as bytes (Serial/TCP) or Modbus frames
- Parsed into JSON or binary format

Step 2: Data Parsing (Worker Thread)

```
Raw Bytes → JSON Parsing → SensorReading Object
```

- Serial/TCP: JSON string parsed into dictionary
- Modbus: Binary frame decoded to 16-bit integer, scaled by 10
- Creates `SensorReading` object with sensor_id, value, timestamp, status

Step 3: Thread-Safe Queue (Worker Thread)

```
SensorReading → Thread-Safe Queue (queue.Queue)
```

- Reading placed in `data_queue` (thread-safe)
- Queue prevents data loss during high-frequency updates
- Multiple sensors can share same queue if on same port/server

Step 4: Callback Registration (Worker Thread)

```
Queue → Callback Function → SensorManager._on_reading_received()
```

- Worker thread calls registered callbacks
- Callbacks executed with thread lock protection
- SensorManager receives reading in callback

Step 5: Alarm Detection (Main Thread)

```
SensorReading → SensorConfig.check_alarm() → AlarmEvent (if triggered)
```

- SensorManager checks value against low_limit/high_limit
- Creates `AlarmEvent` if alarm condition detected
- Includes low_limit and high_limit at time of alarm

Step 6: PyQt Signal Emission (Main Thread)

```
SensorReading → pyqtSignal → GUI Thread Slot
AlarmEvent → pyqtSignal → GUI Thread Slot
```

- `sensor_reading_received.emit(reading)` - Thread-safe signal
- `alarm_triggered.emit(alarm_event)` - Thread-safe signal
- Signals are queued and processed by Qt event loop in GUI thread

Step 7: GUI Update (GUI Thread)

```
Signal Received → on_sensor_reading() → GUI Widgets Updated
```

- `MainWindow.on_sensor_reading()` called in GUI thread
- Updates sensor table, plots, status indicators
- Thread-safe: All GUI operations in main thread only

Data Flow to Web Interface

Step 1: WebSocket Connection

```
Browser → WebSocket Connection → Remote Console Server
```

- Client connects to `ws://localhost:8765`
- Authentication required before commands

Step 2: Data Request (WebSocket Thread)

```
Client Command → Remote Console → SensorManager.get_sensors()
```

- Client sends `get_sensors` command
- Remote Console queries SensorManager for current readings
- Data retrieved from in-memory sensor readings dictionary

Step 3: Data Serialization (WebSocket Thread)

```
SensorReading Objects → JSON Serialization → WebSocket Message
```

- Python objects converted to JSON
- Includes sensor_id, name, value, status, timestamp, unit

Step 4: WebSocket Response (WebSocket Thread)

```
JSON Message → WebSocket Send → Browser Client
```

- Message sent via WebSocket connection
- Client receives real-time updates

Thread Communication Mechanisms

1. PyQt Signals (Thread-Safe) - `pyqtSignal` objects for cross-thread communication - Automatically queued by Qt event loop - Worker threads emit signals, GUI thread receives them - No direct GUI access from worker threads

2. Thread-Safe Queues - `queue.Queue` for data buffering - Prevents data loss during high-frequency updates - Thread-safe put/get operations

3. Thread Locks - `threading.Lock` for shared data protection - Protects sensor_configs dictionary - Prevents race conditions

4. Callback Mechanism - Worker threads call registered callbacks - Callbacks executed with lock protection - SensorManager receives readings via callbacks

Signal Flow Diagram

```
Sensor Simulator
    ↓
Worker Thread (reads port/socket)
    ↓
Parse Data → SensorReading Object
    ↓
Thread-Safe Queue (data_queue.put())
    ↓
Callback Function (with lock)
    ↓
SensorManager._on_reading_received()
    ↓
Alarm Detection (if needed)
    ↓
PyQt Signal Emission (sensor_reading_received.emit())
    ↓
Qt Event Loop (queues signal)
    ↓
GUI Thread Slot (on_sensor_reading())
    ↓
GUI Widgets Updated (table, plots, status)
```

Web Interface Flow Diagram

```
Browser Client
    ↓
WebSocket Connection (ws://localhost:8765)
    ↓
Authentication (username/password)
    ↓
Command: get_sensors
    ↓
Remote Console Handler
    ↓
SensorManager.get_sensors() (reads in-memory data)
    ↓
JSON Serialization
    ↓
WebSocket Send
    ↓
Browser Receives & Displays
```

Key Thread Safety Features

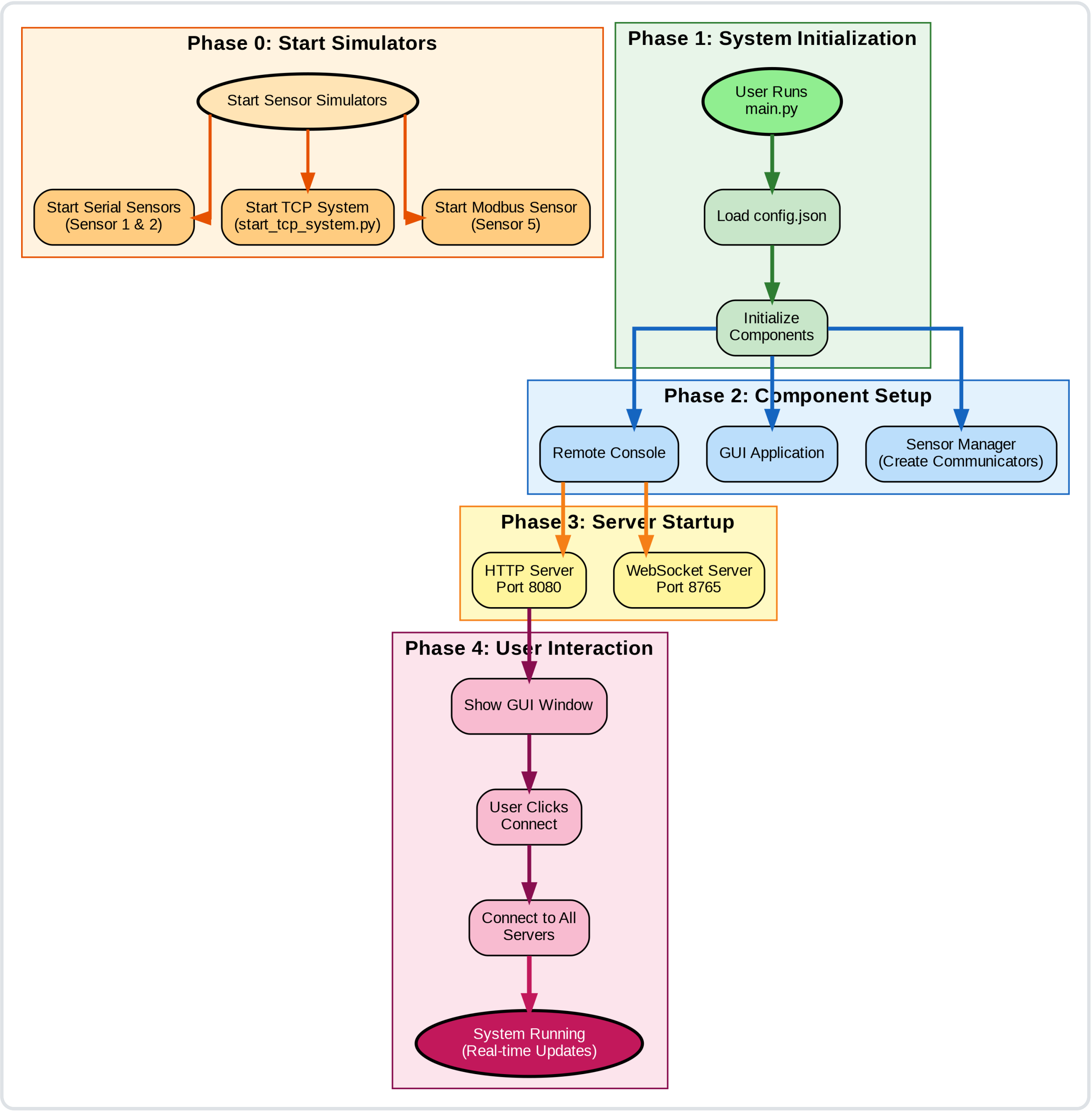
1. No Direct GUI Updates from Workers - Worker threads never directly modify GUI widgets - All GUI updates via PyQt signals - Signals automatically handled in GUI thread

2. Lock Protection - Shared data structures protected with locks - Prevents race conditions - Ensures data consistency

3. Queue-Based Buffering - Thread-safe queues prevent data loss - Handles high-frequency sensor updates - Smooth data flow even under load

4. Signal Queuing - Qt event loop queues signals automatically - Ensures signals processed in correct thread - Prevents GUI freezing

Startup Sequence


Phase 0: Start Simulators
Start Sensor Simulators
Start Serial Sensors (Sensor 1 & 2)
Start TCP System (start_tcp_system.py)
Start Modbus Sensor (Sensor 5)
Phase 1: System Initialization
User Runs main.py
Load config.json
Initialize Components
Phase 2: Component Setup
Remote Console
GUI Application
Sensor Manager (Create Communicators)
Phase 3: Server Startup
HTTP Server Port 8080
WebSocket Server Port 8765
Phase 4: User Interaction
Show GUI Window
User Clicks Connect
Connect to All Servers
System Running (Real-time Updates)

System initialization process: - User launches application - Configuration loading from `config.json` - Sensor Manager initialization - WebSocket and HTTP server startup - GUI window display - Ready for sensor connections

Supported Protocols

Serial Communication

- **Linux**: Pseudo-Terminals (PTY) - `/dev/pts/X`
- **Windows**: COM Ports (real or virtual via com0com)
- JSON frame format, 115200 baudrate

TCP/IP Communication

- TCP socket connections
- Multiple sensors per server
- JSON frame format

Modbus/TCP Communication

- Function Code 3 (Read Holding Registers)
- 16-bit integer encoding (scaled by 10)
- Configurable unit ID and register address

Platform Support

Linux Support

- Full PTY support for serial sensors
- Desktop notifications via `notify-send`
- Native Python 3 support
- Automated startup scripts

Windows Support

- COM port support (real and virtual)
- Desktop notifications via `win10toast`
- COM port pair handling (com0com)
- Automated startup scripts

Cross-Platform Features

- Same codebase for both platforms
- Platform detection and automatic adaptation
- Consistent user experience
- Thread-safe architecture

Key Features

Real-Time Monitoring

- Live sensor data with 2+ Hz refresh rate
- Per-sensor rolling plots (15-second window)
- Color-coded status indicators
- Global system health indicator

Alarm Management

- Automatic LOW/HIGH limit detection
- Faulty sensor detection (-999 values)
- Complete alarm history with limits
- Unlimited notifications (webhook & desktop)

Maintenance Console

- Password-protected access
- Comprehensive alarm log with CSV export
- System diagnostics (self-test, snapshot)
- Live log viewer with color-coded levels

Remote Access

- Web-based interface via browser
- WebSocket API for remote commands
- User authentication and permissions
- Real-time data synchronization

Quick Start Guide

Prerequisites

- Python 3.8 or higher
- Linux (Ubuntu recommended) or Windows
- Internet connection for package installation

Installation Steps

Step 1: Install Dependencies

```
pip3 install -r requirements.txt
```

Step 2: Automated Startup (Recommended)

Linux:

```
./scripts/start_system_linux.sh
```

Windows:

```
scripts\start_system_windows.bat
```

The script automatically: - Starts all sensor simulators - Updates `config.json` with correct ports - Launches the main application

Step 3: Connect Sensors - Click "Connect" button in GUI - Or wait for auto-connect (if configured)

Quick Start Guide (Manual Method)

Step 1: Configure Sensors

Edit `config/config.json` to match your sensor setup: - Sensor IDs, ports, and protocols - Alarm limits (low_limit, high_limit) - Protocol-specific configurations

Step 2: Start Simulators

Modbus Sensors:

```
python3 simulators/sensor_modbus.py --config "voltage:5:localhost:1502:1:0"
```

TCP Sensors:

```
python3 simulators/start_tcp_system.py --server-ports 5000 --sensor flow:3:localhost:5
```

Serial Sensors (Linux):

```
python3 simulators/sensor_serial.py --config "temperature:1:115200:8N1"
```

Serial Sensors (Windows):

```
python simulators\sensor_serial.py --config "temperature:1:115200:8N1" --com-port COM1
```

Step 3: Start Application

```
python3 main.py
```

Step 4: Access Features

- **Desktop**: Monitor dashboard, access maintenance console
- **Web**: Open `http://localhost:8080/remote_console_client.html`

System Components

Core Modules

- `core/sensor_data.py` - Data models (SensorReading, AlarmEvent, SensorConfig)
- `sensors/sensor_manager.py` - Unified sensor management
- `sensors/sensor_*_comm.py` - Protocol-specific communicators

GUI Application

- `gui/main_gui.py` - Main PyQt5 window
- Dashboard with real-time plots and sensor table
- Maintenance console with authentication

Services

- `services/alarm_notifications.py` - Notification system
- `services/remote_console.py` - WebSocket remote console

Simulators

- `simulators/sensor_serial.py` - Serial sensor simulator
- `simulators/start_tcp_system.py` - TCP sensor system
- `simulators/sensor_modbus.py` - Modbus sensor simulator

Threading Architecture

Worker Thread Model

- **One worker thread per unique communication endpoint**
- Multiple sensors on same port/server share one thread
- Thread-safe communication via PyQt signals
- Non-blocking I/O operations

Example Scenarios

- 5 serial sensors on 5 ports = 5 worker threads
- 5 serial sensors on 2 ports = 2 worker threads
- 4 TCP sensors on 2 servers = 2 worker threads
- Mixed setup: Optimized thread usage

Notification System

Desktop Notifications

- **Linux**: `notify-send` command
- **Windows**: `win10toast` library
- Automatic platform detection

Webhook Notifications

- HTTP POST requests (non-blocking)
- Background thread execution
- Configurable URL in `config.json`
- JSON payload with alarm details

Notification Features

- All alarms trigger notifications immediately
- No filtering or state transition checks
- Complete alarm visibility
- Thread-safe implementation

Remote Console API

WebSocket Server

- Port: 8765 (configurable)
- Authentication required
- JSON message format

Available Commands

- `get_status` - System status summary
- `get_sensors` - All sensor readings
- `get_alarms` - Alarm log entries
- `clear_alarms` - Clear alarm log (requires write permission)
- `get_logs` - System log entries
- `run_self_test` - System diagnostics (requires commands permission)
- `get_snapshot` - Detailed system snapshot
- `set_limit` - Set alarm limits (requires write permission)

User Permissions

- **read**: View sensor data and alarms
- **write**: Clear alarms, modify settings
- **commands**: Execute system commands

Configuration

Sensor Configuration

```json
{
  "sensors": [
    {
      "name": "Temperature Sensor 1",
      "id": 1,
      "low_limit": 20.0,
      "high_limit": 80.0,
      "unit": "°C",
      "protocol": "serial",
      "protocol_config": {
        "port": "/dev/pts/9",
        "baudrate": 115200
      }
    }
  ]
}
```

Alarm Settings

```json
{
  "alarm_settings": {
    "enable_notifications": true,
    "enable_desktop_notifications": true,
    "webhook_url": "http://localhost:3000/webhook"
  }
}
```

Remote Console Users

```
{
  "remote_console": {
    "enabled": true,
    "host": "localhost",
    "port": 8765,
    "http_port": 8080,
    "users": {
      "admin": {
        "password": "admin123",
        "permissions": ["read", "write", "commands"]
      }
    }
  }
}
```

Testing

Unit Tests

```
python3 -m pytest tests/
```

Verification

```
python3 scripts/verify_project.py
```

Test Scripts

- `test_websocket.py` - Test WebSocket connection
- `test_modbus.py` - Test Modbus communication
- `test_webhook.py` - Test webhook functionality
- `test_desktop_notifications.py` - Test desktop notifications

Project Structure

```
RT-ProductionLine-Sensors-Dashboard/
├── main.py                    # Main entry point
├── core/                      # Core data models
├── sensors/                   # Sensor communication modules
├── gui/                       # GUI components
├── services/                  # Services (notifications, remote console)
├── simulators/                # Sensor simulators
├── config/                    # Configuration files
├── web/                       # Web interface
├── scripts/                   # Utility scripts
├── tests/                     # Unit tests
└── docs/                      # Documentation
```

Key Benefits

For Operators

- Real-time visibility of all sensors
- Immediate alarm notifications
- Easy-to-use desktop interface
- Quick access to system status

For Administrators

- Comprehensive alarm history
- System diagnostics and snapshots
- Remote access capabilities
- Export functionality (CSV)

For Developers

- Modular, extensible architecture
- Thread-safe implementation
- Cross-platform compatibility
- Well-documented codebase

Technical Highlights

Architecture

- **Modular Design**: Clean separation of concerns
- **OOP**: Object-oriented design throughout
- **Thread-Safe**: Worker threads with PyQt signals
- **Port-Based Workers**: Efficient resource usage

Performance

- 2+ Hz refresh rate for sensor data
- Non-blocking I/O operations
- Optimized thread usage
- Real-time plot updates

Security

- Password-protected maintenance console
- Role-based access control
- WebSocket authentication
- Secure remote access

Platform-Specific Notes

Linux

- PTY devices for serial communication
- Native `notify-send` for desktop notifications
- Full feature support
- Recommended for development

Windows

- COM ports for serial communication
- `win10toast` for desktop notifications
- com0com support for virtual COM ports
- Full feature parity with Linux

Troubleshooting

Common Issues

1. **ModuleNotFoundError**
2. Solution: Run from project root or set `PYTHONPATH=.`
3. **Serial port not found**
4. Solution: Ensure simulators are running and ports match `config.json`
5. **WebSocket connection failed**
6. Solution: Check firewall settings and port availability
7. **No sensor data**
8. Solution: Click "Connect" button and verify simulators are running

Debug Tools

- System Tools → Get Snapshot
- Maintenance Console → Live Log Viewer
- Test scripts in `scripts/` directory

Documentation

Available Documentation

- `README.md` - Quick overview and quick start
- `docs/Project_Documentation.md` - Complete system documentation
- `Project_Documentation.pdf` - PDF version
- `docs/SYSTEM_FLOWCHART.md` - System flowchart documentation
- `docs/WINDOWS_COMPATIBILITY.md` - Windows compatibility guide

Additional Resources

- `tests/README.md` - Unit tests documentation
- `simulators/README.md` - Sensor simulators documentation
- Source code comments for implementation details

Summary

What This System Provides

Real-Time Monitoring - Live sensor data with 2+ Hz refresh rate
Multi-Protocol Support - Serial, TCP/IP, and Modbus/TCP
Intelligent Alarms - Automatic detection with complete history
Cross-Platform - Full Linux and Windows support
Remote Access - Web-based interface with authentication
Professional UI - Modern desktop and web interfaces
Comprehensive Logging - System logs and alarm history
Extensible Architecture - Modular, thread-safe design

Thank You

Real-Time Production Line Sensor Dashboard

A comprehensive solution for industrial production line monitoring and maintenance.

Developed as part of the Si-Ware System Assessment

This project demonstrates: - Advanced multi-protocol sensor communication - Cross-platform compatibility (Linux/Windows) - Thread-safe architecture with real-time updates - Professional desktop and web interfaces - Comprehensive alarm management system

For more information, visit the project documentation.

Last Updated: January 2026