AsyncIO and PyQt6 Integration Architecture

Architecture Diagrams

Document Version: 1.0 **Last Updated:** 2025-10-31 **Status:** Week 5 - Research Mode Roadmap

Overview

TOSCA uses a safe asyncio integration pattern to execute treatment protocols asynchronously while maintaining thread safety with PyQt6's event loop. This document explains the architecture, patterns, and best practices.

Table of Contents

Architecture Overview

Key Components

```
                    Main GUI Thread
               (PyQt6 Event Loop)


     MainWindow            ► ProtocolWorker
      (UI)                     (QRunnable)

                                      Submit to ThreadPool


                                ▼

                QThreadPool Worker Thread


       asyncio.run() creates event loop


          ProtocolEngine
          execute_protocol()

          async def action_1()
          await asyncio.sleep()
          async def action_2()


       Event loop destroyed on completion


   Signals (thread-safe)                       ► Main GUI
```

Design Philosophy

1. **Single Event Loop per Thread:** Each worker thread has its own asyncio event loop
2. **Isolated Execution:** Event loops never conflict with PyQt6's main event loop

3. **Signal-Based Communication:** Qt signals provide thread-safe communication
4. **Proper Lifecycle Management:** Event loops created and destroyed cleanly

---

Threading Model

Thread Types

Thread	Purpose	Event Loop	Lifetime
Main GUI Thread	Qt event loop, UI rendering, user interaction	PyQt6 QEventLoop	Application lifetime
Worker Threads	Protocol execution, async operations	asyncio event loop	Task duration only

Thread Safety Guarantees

1. **Hardware Controllers:** Use `threading.RLock` for thread-safe operations
2. **Qt Signals:** Automatically queued across thread boundaries
3. **Cancel Events:** `threading.Event` for safe cancellation
4. **No Shared State:** Worker threads operate on isolated data

---

Event Loop Lifecycle

Worker Thread Event Loop

```python
def run(self) -> None:
    """
    Execute protocol in background thread.

    Event loop lifecycle:
    1. Created by asyncio.run()
    2. Used for async protocol execution
    3. Automatically destroyed on completion
    """
    # asyncio.run() handles entire lifecycle
    success, message = asyncio.run(
        self.protocol_engine.execute_protocol(
            self.protocol,
            record=True,
            cancel_event=self._cancel_event,
        )
    )
```

Lifecycle Phases

```
Phase 1: Thread Start
 - QThreadPool assigns worker to thread
 - Worker.run() method called

                 ▼

Phase 2: Event Loop Creation
 - asyncio.run() creates new event loop
 - Loop set as current event loop for this thread
 - No interaction with Qt's main event loop

                 ▼

Phase 3: Async Execution
```

```
  - Protocol engine executes async actions
  - await statements coordinate asynchronous operations
  - asyncio.sleep() for non-blocking delays


               ▼

 Phase 4: Completion & Cleanup
  - asyncio.run() waits for protocol completion
  - Event loop closed and destroyed
  - Worker thread returns to thread pool
```

Safe Pattern: QRunnable + asyncio.run()

Why This Pattern is Safe

```python
class ProtocolWorker(QRunnable):
    """SAFE: Creates isolated event loop in worker thread."""

    def run(self) -> None:
        # SAFE: asyncio.run() creates temporary event loop
        # that exists ONLY in this thread and ONLY during execution.
        # No conflict with Qt's main event loop.
        success, message = asyncio.run(
            self.protocol_engine.execute_protocol(self.protocol)
        )
        self.signals.execution_complete.emit(success, message)
```

Safety Properties

1. **Event Loop Isolation**
   - Event loop created fresh for each execution
   - Exists entirely within worker thread
   - No shared state with Qt's event loop
2. **Clean Lifecycle**
   - `asyncio.run()` handles loop creation
   - Automatic cleanup on completion/exception
   - No dangling event loops
3. **Thread-Safe Communication**
   - Qt signals automatically queued
   - Main thread receives signals safely
   - No direct cross-thread calls

Dangerous Pattern: asyncio-in-QThread (AVOIDED)

What NOT to Do

```python
# DANGEROUS: DO NOT USE THIS PATTERN!
class DangerousAsyncQThread(QThread):
    def run(self) -> None:
        # DANGEROUS: Persistent event loop in QThread
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)

        try:
            # This loop persists for thread lifetime
            loop.run_forever()  # DANGEROUS!
        finally:
            loop.close()
```

Why This is Dangerous

Problem	Description
Event Loop Conflicts	Qt and asyncio event loops can interfere
Complex Lifecycle	Manual loop management error-prone
Shutdown Issues	Difficult to cleanly shut down running loop
Resource Leaks	Loops may not close properly on errors
Debugging Nightmare	Race conditions hard to reproduce

TOSCA's Safe Alternative

```
# SAFE: QRunnable with asyncio.run()
class ProtocolWorker(QRunnable):
    def run(self) -> None:
        # Event loop created, used, destroyed - all automatic
        asyncio.run(self.async_work())
```

Protocol Execution Flow

High-Level Flow

```
1. **User clicks "Start Protocol" button in UI**
2. **MainWindow creates ProtocolWorker(protocol_engine, protocol)**
3. **Worker submitted to QThreadPool**
4. **Thread pool assigns worker to available thread**
5. **Worker.run() executes in background thread** -
6. **a. asyncio.run() creates event loop**
7. **b. protocol_engine.execute_protocol() called**
8. **c. Async actions executed sequentially** -
9. **- await self._execute_set_laser_power()**
10. **- await asyncio.sleep(duration)**
11. **- await self._execute_move_actuator()**
12. **d. Event loop automatically closed**
13. **Signals emitted to main thread** -
14. **- execution_started.emit()**
15. **- progress_update.emit(percentage, status)**
16. **- execution_complete.emit(success, message)**
17. **Main thread updates UI in response to signals**
```

Async Execution Within Protocol

```
async def execute_protocol(self, protocol: Protocol) -> tuple[bool, str]:
    """Async protocol execution in worker thread event loop."""

    for action in protocol.actions:
        # Each action is async and can use await
        await self._execute_action(action)

        # Non-blocking delays
        if action.delay_after:
            await asyncio.sleep(action.delay_after)

        # Check for pause/cancel
        await self._pause_event.wait()
        if self._stop_requested:
            break

    return True, "Protocol completed"
```

Signal/Slot Integration

Signal Emission from Worker Thread

```
class ProtocolWorker(QRunnable):
    def run(self) -> None:
```

```python
        # Signals emitted from worker thread
        self.signals.execution_started.emit()  # Thread-safe!

        try:
            success, message = asyncio.run(...)
            self.signals.execution_complete.emit(success, message)
        except Exception as e:
            self.signals.execution_error.emit(str(e))
```

Signal Reception in Main Thread

```python
class MainWindow(QMainWindow):
    def start_protocol(self):
        worker = ProtocolWorker(self.protocol_engine, protocol)

        # Connect signals (automatically queued across threads)
        worker.signals.execution_started.connect(self._on_execution_started)
        worker.signals.execution_complete.connect(self._on_execution_complete)
        worker.signals.progress_update.connect(self._on_progress_update)

        # Submit to thread pool
        QThreadPool.globalInstance().start(worker)

    def _on_execution_started(self):
        """Called in main thread via Qt's queued connection."""
        self.status_label.setText("Protocol running...")

    def _on_progress_update(self, percentage: int, status: str):
        """Called in main thread."""
        self.progress_bar.setValue(percentage)
```

Connection Types

Type	When Used	Thread Safety
Queued	Signal emitted from worker thread to main thread	Thread-safe (default for cross-thread)
Direct	Signal emitted and received in same thread	Not thread-safe across threads
Auto	Qt determines based on thread	Safe (chooses queued for cross-thread)

Error Handling and Cancellation

Exception Handling

```python
class ProtocolWorker(QRunnable):
    def run(self) -> None:
        try:
            # asyncio.run() handles async exceptions
            success, message = asyncio.run(
                self.protocol_engine.execute_protocol(...)
            )
            self.signals.execution_complete.emit(success, message)

        except asyncio.CancelledError:
            # Handle async cancellation
            self.signals.execution_complete.emit(False, "Cancelled")

        except Exception as e:
            # Catch all other exceptions
            logger.error(f"Protocol error: {e}", exc_info=True)
            self.signals.execution_error.emit(str(e))
```

Cooperative Cancellation

```python
# GUI Thread: Request cancellation
def cancel_protocol(self):
```

```python
    self.current_worker.cancel()  # Thread-safe

# Worker Thread: Check cancellation
class ProtocolWorker(QRunnable):
    def __init__(self):
        self._cancel_event = threading.Event()  # Thread-safe

    def cancel(self):
        self._cancel_event.set()  # Can be called from any thread

    def run(self):
        asyncio.run(
            self.protocol_engine.execute_protocol(
                cancel_event=self._cancel_event  # Pass to async code
            )
        )

# Protocol Engine: Respect cancellation
async def execute_protocol(self, cancel_event):
    for action in actions:
        if cancel_event.is_set():
            return False, "Cancelled"
        await self._execute_action(action)
```

Best Practices

1. Always Use QRunnable for Async Work

```python
# GOOD: QRunnable + asyncio.run()
class MyWorker(QRunnable):
    def run(self):
        asyncio.run(self.async_task())

# BAD: QThread with persistent event loop
class BadWorker(QThread):
    def run(self):
        loop = asyncio.new_event_loop()  # DON'T!
        loop.run_forever()
```

2. Use Signals for Cross-Thread Communication

```python
# GOOD: Signal emission
worker.signals.progress.emit(percentage)

# BAD: Direct GUI update from worker
self.ui.progress_bar.setValue(percentage)  # Thread-unsafe!
```

3. Handle Event Loop Lifecycle Automatically

```python
# GOOD: asyncio.run() manages everything
def run(self):
    asyncio.run(self.async_work())

# BAD: Manual loop management
def run(self):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    try:
        loop.run_until_complete(self.async_work())
    finally:
        loop.close()  # Easy to forget!
```

4. Use threading.Event for Cancellation

```python
# GOOD: Thread-safe Event
self._cancel_event = threading.Event()

# BAD: Simple boolean flag
self._cancelled = False  # Race conditions!
```

5. Never Block the Main Thread

```python
# GOOD: Async work in worker thread
worker = ProtocolWorker(...)
QThreadPool.globalInstance().start(worker)

# BAD: Blocking main thread
asyncio.run(long_running_task())  # GUI freezes!
```

Implementation Examples

Example 1: Simple Async Task

```python
from PyQt6.QtCore import QRunnable, QObject, pyqtSignal
import asyncio

class SimpleWorkerSignals(QObject):
    finished = pyqtSignal(str)

class SimpleWorker(QRunnable):
    def __init__(self, duration: float):
        super().__init__()
        self.duration = duration
        self.signals = SimpleWorkerSignals()

    async def async_task(self):
        await asyncio.sleep(self.duration)
        return f"Completed after {self.duration}s"

    def run(self):
        result = asyncio.run(self.async_task())
        self.signals.finished.emit(result)
```

Example 2: Progress Reporting

```python
class ProgressWorkerSignals(QObject):
    progress = pyqtSignal(int, str)
    finished = pyqtSignal()

class ProgressWorker(QRunnable):
    def __init__(self, steps: int):
        super().__init__()
        self.steps = steps
        self.signals = ProgressWorkerSignals()

    async def async_work(self):
        for i in range(self.steps):
            percentage = int((i + 1) / self.steps * 100)
            status = f"Step {i+1}/{self.steps}"

            # Emit signal (thread-safe)
            self.signals.progress.emit(percentage, status)

            # Async delay
            await asyncio.sleep(0.5)

    def run(self):
        asyncio.run(self.async_work())
        self.signals.finished.emit()
```

Example 3: Cancellable Task

```python
import threading

class CancellableWorkerSignals(QObject):
    finished = pyqtSignal(bool, str)

class CancellableWorker(QRunnable):
    def __init__(self):
```

```python
        super().__init__()
        self.signals = CancellableWorkerSignals()
        self._cancel_event = threading.Event()

    async def async_work(self):
        for i in range(100):
            if self._cancel_event.is_set():
                return False, "Cancelled"

            # Do work
            await asyncio.sleep(0.1)

        return True, "Completed"

    def run(self):
        success, message = asyncio.run(self.async_work())
        self.signals.finished.emit(success, message)

    def cancel(self):
        """Thread-safe cancellation."""
        self._cancel_event.set()
```

Comparison with Other Patterns

Pattern Comparison Table

Pattern	Event Loop Location	Lifecycle	Thread Safety	Recommended
QRunnable + asyncio.run()	Worker thread	Automatic	High	✅ YES (TOSCA uses this)
asyncio-in-QThread	QThread	Manual	Medium	❌ NO (dangerous)
qasync library	Main thread	Shared Qt/asyncio	Medium	⚠️ Maybe (complex)
Pure threading	N/A	Manual	Low	❌ NO (no async support)

Troubleshooting

Common Issues

Issue: “RuntimeError: There is no current event loop in thread”

Cause: Trying to use `asyncio.get_event_loop()` in worker thread

Solution: Use `asyncio.run()` which creates the loop automatically

```python
# GOOD
def run(self):
    asyncio.run(self.async_work())

# BAD
def run(self):
    loop = asyncio.get_event_loop()  # Fails in worker thread!
```

Issue: GUI freezes during protocol execution

Cause: Running async code in main thread

Solution: Always use QRunnable worker threads

```python
# GOOD: Non-blocking
worker = ProtocolWorker(...)
```

```python
QThreadPool.globalInstance().start(worker)

# BAD: Blocks main thread
asyncio.run(protocol.execute())  # DON'T in main thread!
```

Issue: Signals not received in main thread

Cause: Forgot to inherit from QObject for signals class

Solution: Signals class must inherit QObject

```python
# GOOD
class WorkerSignals(QObject):
    finished = pyqtSignal()

# BAD
class WorkerSignals:  # Missing QObject!
    finished = pyqtSignal()
```

References

Related Documentation

- 10_concurrency_model.md - Threading patterns and RLock usage
- 01_system_overview.md - Overall architecture
- 04_treatment_protocols.md - Protocol execution details

External Resources

- Qt Documentation: QThreadPool
- Python asyncio Documentation
- Qt Thread Basics

Source Code

- `src/ui/workers/protocol_worker.py` - QRunnable worker implementation
- `src/core/protocol_engine.py` - Async protocol execution engine

Document Status: Complete **Review Date:** 2025-11-30 **Next Review:** Upon protocol execution changes