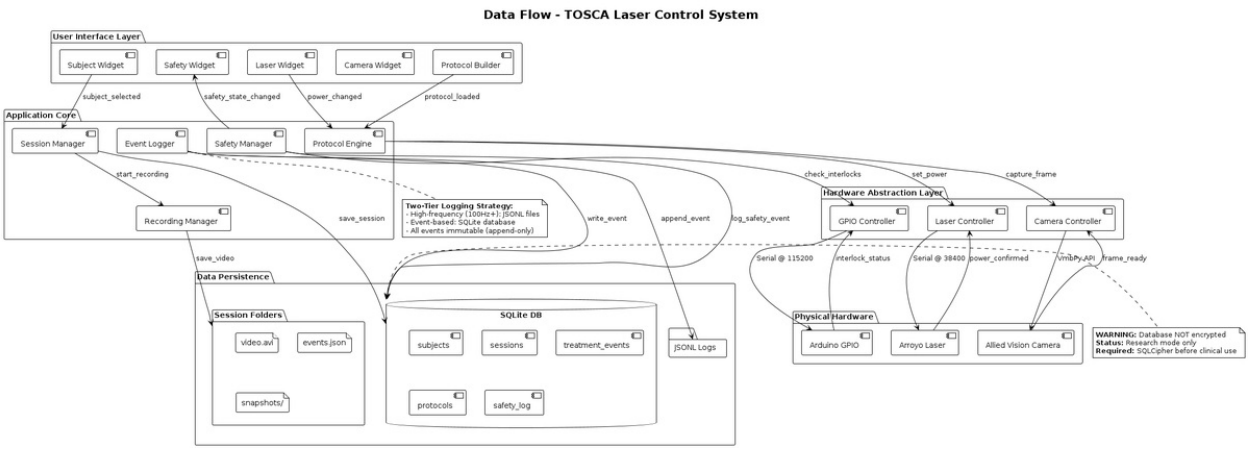
Concurrency & Threading Model

Architecture Diagrams

Figure 1: TOSCA Data Flow Diagram



TOSCA Data Flow Diagram

Document Version: 1.0 **Last Updated:** 2025-10-26 **Status:** Implemented - Phase 5 **Priority:** CRITICAL - Thread safety required for medical device

---

Table of Contents

Overview

Purpose

This document describes the threading and concurrency model for TOSCA, including thread ownership, communication patterns, and thread-safety guarantees.

Why Threading is Critical: - **Real-time camera streaming** (60+ FPS) cannot block GUI - **Hardware I/O operations** (laser serial, GPIO) have delays - **Safety watchdog** requires continuous heartbeat - **Database operations** cannot freeze UI during logging

Threading Philosophy

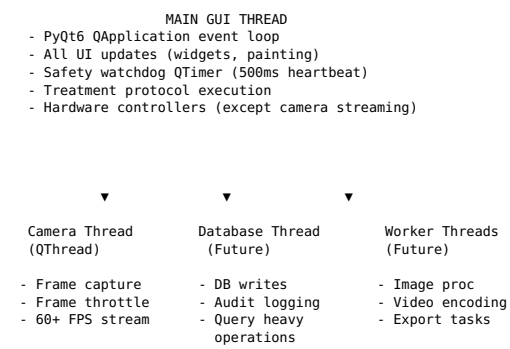
Primary Principle: PyQt6 signals/slots for all cross-thread communication

Benefits: - [DONE] Thread-safe by design (Qt event loop handles synchronization) - [DONE] Automatic queuing (signals queued across thread boundaries) - [DONE] Type-safe (pyqtSignal validates parameter types) - [DONE] Decoupled (emitters don't know about receivers)

Avoid: - [FAILED] Manual mutex/lock management (error-prone) - [FAILED] Shared mutable state (race conditions) - [FAILED] Direct cross-thread method calls (unsafe)

Threading Architecture

Primary Threads



Thread Breakdown

Thread	Created By	Lifespan	Purpose
Main GUI	QApplication	App lifetime	UI updates, user input, main event loop
Camera Stream	QThread	Camera active	60 FPS frame capture and emission
Database	(Future)	On-demand	Heavy database queries, export operations
Workers	(Future)	Task duration	Image processing, video encoding

Current Status (Phase 5): - [DONE] Main GUI thread operational - [DONE] Camera stream thread implemented (QThread) - [PENDING] Database thread planned (Phase 6) - [PENDING] Worker pool planned (Phase 7+)

Thread Ownership

Main GUI Thread Owns

Core Business Logic: - `SafetyManager` - Safety state, interlock validation - `ProtocolEngine` - Treatment protocol execution - `SessionManager` - Session lifecycle management

Hardware Controllers (except camera streaming): - `LaserController` - Arroyo laser serial communication - `ActuatorController` - Xeryon actuator control - `GPIOController` - Arduino Nano serial communication

UI Widgets: - All `QWidget` subclasses (thread affinity to main thread) - `MainWindow`, `TreatmentWidget`, `SafetyWidget`, etc.

Timers: - `SafetyWatchdog` QTimer (500ms heartbeat) - UI update timers (status polling, etc.)

Camera Thread Owns

File: `src/hardware/camera_controller.py`

Class: `CameraStreamThread(QThread)`

Responsibilities: - Frame capture from Allied Vision camera (VmbPy SDK) - Frame throttling (60 FPS camera → 30 FPS GUI) - Frame format conversion (Bayer → BGR via numpy) - Signal emission: `frame_ready`, `fps_update`, `error_occurred`

Ownership: - `vmbpy.Camera` instance (camera handle) - Frame buffers (numpy arrays) - FPS calculation state

Thread-Safety: - [DONE] No shared mutable state with main thread - [DONE] Communicates via PyQt6 signals only - [DONE] Self-contained (no callbacks from main thread)

Database Thread (Future - Phase 6)

Planned Responsibilities: - Heavy SELECT queries (treatment history, audit export) - Bulk INSERT operations (session logging) - Database vacuum/optimization - Report generation

Why Separate Thread: - Database operations can take 100-500ms - Would freeze GUI if run on main thread - Non-critical for real-time operation

---

Inter-Thread Communication

PyQt6 Signal/Slot Pattern

Core Mechanism:

```
1. **# In camera thread**
2. **class CameraStreamThread(QThread)** -
3. **frame_ready = pyqtSignal(np.ndarray)  # Signal definition**
4. **def run(self)** -
5. **"""Thread execution (in camera thread)."""**
6. **while self.running** -
7. **frame = capture_frame()**
8. **self.frame_ready.emit(frame)  #  Emit signal (thread-safe)**
9. **# In main thread**
10. **class CameraWidget(QWidget)** -
11. **def __init__(self)** -
12. **super().__init__()**
13. **self.stream_thread = CameraStreamThread(camera)**
14. **# Connect signal to slot (automatic queuing across threads)**
15. **self.stream_thread.frame_ready.connect(self.update_frame)**
16. **#                        Slot runs in MAIN thread (safe)**
17. **def update_frame(self, frame** - np.ndarray):
18. **"""Update UI with frame (runs in main thread)."""**
19. **pixmap = convert_to_pixmap(frame)**
20. **self.image_label.setPixmap(pixmap)  #  UI update (safe)**
```

What Happens Under the Hood: 1. `frame_ready.emit(frame)` called in camera thread 2. Qt detects signal crosses thread boundary 3. Signal **automatically queued** to main thread event loop 4. Main thread processes event queue 5. `update_frame(frame)` called in main thread 6. UI updated safely (all UI updates in main thread)

Benefits: - [DONE] No manual locking required - [DONE] Automatic memory management (Qt copies signal data) - [DONE] Type-safe (pyqtSignal enforces parameter types) - [DONE] Decoupled (emitter doesn't know about receiver)

Frame Throttling Example

Problem: Camera captures 60 FPS, but GUI only needs 30 FPS (reduces CPU load)

Solution:

```
class CameraStreamThread(QThread):
    def __init__(self, camera):
        super().__init__()
        self.gui_fps_target = 30.0
        self.last_gui_frame_time = 0.0

    def run(self):
        """Throttle frames to 30 FPS for GUI."""
        while self.running:
            frame = camera.get_frame()  # 60 FPS capture

            current_time = time.time()
            time_since_last = current_time - self.last_gui_frame_time
            min_interval = 1.0 / self.gui_fps_target  # 33ms for 30 FPS

            if time_since_last >= min_interval:
                self.frame_ready.emit(frame)  # Only emit every 33ms
                self.last_gui_frame_time = current_time
```

Result: Camera runs at 60 FPS (smooth capture), GUI updates at 30 FPS (responsive UI, lower CPU)

Signal Connection Types

Qt::AutoConnection (Default): - Same thread → Direct call (synchronous) - Different threads → Queued call (asynchronous) - PyQt6 chooses automatically based on thread affinity

Qt::DirectConnection: - Always direct call (synchronous) - WARNING: UNSAFE across threads (can cause race conditions) - Only use within same thread

Qt::QueuedConnection: - Always queued (asynchronous) - Safe across threads - Slight latency (event loop processing)

Example:

```
# Default (recommended)
signal.connect(slot)  # Auto connection

# Explicit queued (cross-thread safety)
signal.connect(slot, Qt.ConnectionType.QueuedConnection)
```

Thread Safety Guarantees

Hardware Controllers

LaserController, ActuatorController, GPIOController:

Thread Affinity: Main GUI thread

Thread Safety: - [DONE] All methods called from main thread only - [DONE] Serial I/O operations (pyserial) not thread-safe - [DONE] PyQt6 signals emitted in main thread

No Locking Required: - All hardware operations sequential in main thread - No concurrent access possible

CameraController

Thread Affinity: Mixed (main thread for control, camera thread for streaming)

Thread Safety Strategy:

```
class CameraController(HardwareControllerBase):
    def __init__(self):
        super().__init__()
        self.stream_thread = CameraStreamThread(camera)
        self._recording = False  #  Accessed from main thread only

    # Main thread methods
    def start_streaming(self):
        """Called from main thread."""
        self.stream_thread.start()  #  Starts camera thread

    def stop_streaming(self):
        """Called from main thread."""
        self.stream_thread.running = False  #  Signal to stop
        self.stream_thread.wait()  #  Wait for thread to finish

    # Camera thread emits signals
    # Main thread receives signals in slots
    #  No shared mutable state, no locking needed
```

Key Insight: Camera thread is **self-contained** and communicates via signals only.

SafetyManager

Thread Affinity: Main GUI thread

Thread Safety: - [DONE] All safety logic runs in main thread - [DONE] Watchdog heartbeat (QTimer) runs in main thread - [DONE] No concurrent access to safety state

Critical Section: None (single-threaded)

Database Operations (Current - Phase 5)

Thread Affinity: Main GUI thread

Thread Safety: - [DONE] SQLite database operations are thread-safe (serialized mode) - [DONE] All database calls from main thread only - WARNING: Future: Move heavy queries to separate thread (Phase 6)

Concurrency Patterns

Pattern 1: QThread for Long-Running Tasks

Use Case: Camera streaming (continuous 60 FPS capture)

Implementation:

```
class WorkerThread(QThread):
    result_ready = pyqtSignal(object)

    def run(self):
        """Override run() for thread execution."""
        while self.running:
            result = do_heavy_computation()
            self.result_ready.emit(result)
```

Benefits: - [DONE] Keeps GUI responsive - [DONE] Easy to start/stop (QThread API) - [DONE] Automatic signal queuing

Pattern 2: QTimer for Periodic Tasks

Use Case: Safety watchdog heartbeat (500ms interval)

Implementation:

```
class SafetyWatchdog(QObject):
    def __init__(self):
        super().__init__()
        self.timer = QTimer()
        self.timer.setInterval(500)  # 500ms
        self.timer.timeout.connect(self._send_heartbeat)

    def start(self):
        self.timer.start()  # ← Runs in main thread event loop

    def _send_heartbeat(self):
        """Called every 500ms (main thread)."""
        self.gpio_controller.send_watchdog_heartbeat()
```

Benefits: - [DONE] No separate thread needed - [DONE] Runs in main thread (safe for UI updates) - [DONE] Automatic Qt integration

Pattern 3: Signals/Slots for Decoupling

Use Case: Hardware events trigger UI updates

Implementation:

```
# Hardware controller
class LaserController(HardwareControllerBase):
    power_changed = pyqtSignal(float)

    def set_power(self, power: float):
        self._send_serial_command(f"POWER {power}")
        self.power_changed.emit(power)  # ← Notify listeners
```

```python
# UI widget
class LaserWidget(QWidget):
    def __init__(self, laser: LaserController):
        super().__init__()
        laser.power_changed.connect(self.update_power_display)

    def update_power_display(self, power: float):
        self.power_label.setText(f"{power:.1f} W")
```

Benefits: - [DONE] Decoupled (hardware doesn't know about UI) - [DONE] Thread-safe (automatic queuing) - [DONE] Testable (can mock signal connections)

Deadlock Prevention

No Manual Locking (Current Design)

Current Status: No mutexes or locks in codebase

Why: - PyQt6 signals handle all synchronization - Camera thread is self-contained (no shared state) - All hardware I/O in main thread (no concurrent access)

Benefits: - [DONE] No deadlock possible (no locks) - [DONE] Simpler code (less error-prone) - [DONE] Easier to reason about

Future Considerations (Phase 6+)

If Adding Manual Locks:

Rule 1: Lock Ordering

```python
# Always acquire locks in same order
lock_a.acquire()
lock_b.acquire()
# ... critical section ...
lock_b.release()
lock_a.release()
```

Rule 2: Lock Timeout

```python
# Use timeout to detect deadlock
if lock.acquire(timeout=1.0):
    try:
        # ... critical section ...
    finally:
        lock.release()
else:
    logger.error("Lock timeout - possible deadlock")
```

Rule 3: Prefer Signals/Slots

```python
# GOOD: Use signals (no locks needed)
self.data_ready.emit(data)

# BAD: Manual locking (error-prone)
with self.lock:
    self.shared_data = data
```

Testing Concurrency

Thread Safety Tests

Test 1: Verify Signal Thread Affinity

```python
def test_camera_signal_thread_affinity():
    """Verify frame_ready signal runs in main thread."""
    main_thread_id = threading.current_thread().ident
    received_thread_id = None

    def capture_thread_id(frame):
        nonlocal received_thread_id
        received_thread_id = threading.current_thread().ident

    camera.frame_ready.connect(capture_thread_id)
    camera.start_streaming()
    time.sleep(0.1)  # Let frame arrive

    assert received_thread_id == main_thread_id
```

Test 2: Verify No Shared State

```python
def test_camera_thread_isolation():
    """Verify camera thread has no shared mutable state."""
    # Camera thread should only emit signals
    # Should not modify main thread state directly
    assert not hasattr(camera.stream_thread, 'parent_ref')
```

Performance Tests

Test Frame Throttling:

```python
def test_frame_throttling():
    """Verify GUI FPS is limited to 30 FPS."""
    frame_times = []

    def record_time(frame):
        frame_times.append(time.time())

    camera.frame_ready.connect(record_time)
    camera.start_streaming()
    time.sleep(1.0)
```

```
camera.stop_streaming()

# Verify FPS is ~30, not 60
fps = len(frame_times) / 1.0
assert 28 <= fps <= 32  # Allow ±2 FPS tolerance
```

Best Practices

Do's ✅

1. **Use PyQt6 signals for cross-thread communication**
2. **Keep camera thread self-contained** (no shared state)
3. **Run all UI updates in main thread**
4. **Use QTimer for periodic tasks** (instead of separate thread)
5. **Document thread ownership** (which thread owns what)

Don'ts ❌

1. **Don't use manual locks** (prefer signals/slots)
2. **Don't share mutable state** between threads
3. **Don't call UI methods from worker threads** (use signals)
4. **Don't block main thread** (move long tasks to QThread)
5. **Don't use threading.Thread** (use QThread for Qt integration)

References

PyQt6 Threading

- **Qt Threading Basics:** https://doc.qt.io/qt-6/threads-technologies.html
- **QThread Class:** https://doc.qt.io/qt-6/qthread.html
- **Signals & Slots:** https://doc.qt.io/qt-6/signalsandslots.html

Thread Safety

- **Python GIL:** https://docs.python.org/3/glossary.html#term-global-interpreter-lock
- **Thread-Safe Queue:** https://docs.python.org/3/library/queue.html

---

Document Owner: Software Architect **Last Updated:** 2025-10-26 **Next Review:** Before Phase 6 (database threading) **Status:** Implemented - Current threading model documented