# Milestone 3.2: Device Connection Manager and Socket Server

## Overview and Goals of Milestone 3.2

In this milestone, we will implement the **Device Connection Manager and Socket Server** for the PC application. The goal is to enable the PC app to act as a network server that accepts connections from the Android devices and manages their state within the UI. This involves setting up a TCP socket server (listening on a specified port, e.g. 9000), handling multiple device connections (using multi-threading for concurrency), and updating the GUI to reflect the status of each connected device. By the end of this milestone, the PC app will function as the orchestrator of the system, capable of communicating with one or more Android devices in real-time.

**Key objectives include:**

* **Socket Server Setup:** Creating a TCP server on the PC that listens for incoming device connections (on port 9000, matching the port used by the Android clients).
* **Device Registration:** When a device connects, receiving an initial handshake (a JSON "hello" message with device ID and capabilities), registering the device in the application (creating a RemoteDevice object), and updating the UI (e.g. adding the device to a device list panel with its status and capabilities).
* **Message Handling:** Implementing a loop to continually receive and parse JSON messages from each connected device. Depending on message type, update the UI (e.g. display preview images, update battery status, log acknowledgments, etc.).
* **Outgoing Commands:** Providing a way for the PC app to send JSON commands to one or all connected devices (for example, to start/stop recording) in a thread-safe manner.
* **Connection Management:** Gracefully handling device disconnections (updating UI status, allowing reconnections) and ensuring the server remains running for new connections.
* **Testing:** Verifying the communication flow using simulated devices or test scripts, ensuring that devices can connect, send data, receive commands, and handle disconnects properly.

Throughout this guide, we will break down the implementation steps, propose a structure for classes/modules, discuss how to configure the development environment (IDE) for this component, and outline tests to validate that the device connection manager works as expected.

## System Architecture for Device Connections

Before diving into implementation, it's important to understand how the pieces fit together:

* **PC Application (Server):** Runs a socket server that listens on a known port (e.g. 9000) for incoming TCP connections. It will use Python sockets in a multi-threaded setup: one thread (the "listening thread") accepts new connections, and for each device connection, a new "device handler" thread is spawned to manage communication with that device. The PC app will maintain a list (or dictionary) of active devices and their states. The PC is responsible for sending control commands to devices and aggregating data from them.
* **Android Devices (Clients):** Each device runs a client that knows the server’s IP address and port. On connection, it sends a handshake JSON (with its identifier and available sensor/camera capabilities). Subsequently, the device continuously sends data messages (e.g., preview frames, sensor readings, status updates) as JSON objects, and listens for commands from the server.
* **Communication Protocol:** We use a simple text-based protocol with JSON messages. Each message can be a standalone JSON object (we can delineate messages by newline \n or a special delimiter, or use a length-prefixed protocol). For simplicity, we assume each JSON message is sent followed by a newline, so the server can read and split incoming data on newline boundaries to reconstruct each JSON message. Example message types:
* **hello** – sent by device on connect with device ID and capabilities.
* **preview\_frame** – contains a base64-encoded image from the device’s camera preview.
* **status\_update** – contains status info like battery level, storage, etc.
* **sensor\_data** – contains readings from sensors (if sent live).
* **ack / error** – acknowledgements or error responses to commands.
* **notification** – e.g. a message indicating an event (like "recording\_finished").
* **UI Integration:** The PC app’s GUI (likely built with PyQt/PySide) will display a list of devices and their status. For each connected device, the UI might show:
* Device ID or name.
* Icons or indicators for capabilities (e.g. camera available, thermal camera, Shimmer sensor, etc.).
* Live preview image (updated as preview\_frame messages arrive).
* Status info like battery percentage or connection status.

The GUI will also have controls (buttons) to send commands (e.g. "Start Recording", "Stop Recording") to devices. These controls will invoke functions that send JSON commands via the socket server to the devices.

Given this architecture, let's proceed step-by-step to implement the Device Connection Manager and Socket Server.

## Step-by-Step Implementation Guide

### 1. Setting Up the Socket Server (Listening for Connections)

**Goal:** Initialize a TCP server socket in the PC app that listens for incoming device connections on a specified port (port **9000** in our case). This server should run without freezing the GUI, so it must be run on a separate thread or use asynchronous I/O.

**Implementation Steps:**

* **Choose a Port:** Decide on the port number (9000 is suggested). Ensure this port is not blocked by firewall or already in use. Use a constant or configuration for easy change.
* **Create a Server Socket:** In Python, use the socket module to create a socket. For TCP:
* import socket  
  server\_sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
  server\_sock.bind(('0.0.0.0', 9000)) # listen on all network interfaces  
  server\_sock.listen(5) # allow up to 5 queued connections  
  print("Server listening on port 9000")
* This binds to all IPs of the host (so devices on the local network can connect) and starts listening[[1]](https://www.geeksforgeeks.org/python/socket-programming-multi-threading-python/#:~:text=host%20%3D%20%27%27%20port%20%3D,port). The backlog of 5 is usually fine (max 5 pending connections).
* **Run in a Background Thread:** To avoid blocking the main UI thread, start the server in a separate thread. For example, create a class DeviceServerThread that extends threading.Thread:
* import threading  
  class DeviceServerThread(threading.Thread):  
   def \_\_init\_\_(self, host='0.0.0.0', port=9000):  
   super().\_\_init\_\_(daemon=True) # daemon=True so it won't prevent app exit  
   self.host = host  
   self.port = port  
   self.server\_sock = None  
   self.running = False  
    
   def run(self):  
   # Initialize socket and listen  
   self.server\_sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
   self.server\_sock.bind((self.host, self.port))  
   self.server\_sock.listen(5)  
   self.running = True  
   print(f"Server listening on {self.host}:{self.port}")  
   # Accept loop  
   while self.running:  
   try:  
   client\_sock, client\_addr = self.server\_sock.accept()  
   except OSError:  
   # Socket was closed or error  
   break  
   print(f"Accepted connection from {client\_addr}")  
   # Handle the new connection (see next steps)  
   self.handle\_new\_client(client\_sock, client\_addr)  
   print("Server thread terminating.")
* In the above, we mark the thread as daemon for safety (so it won’t hang the app on exit, though we'll also handle graceful shutdown). The accept() call will block until a client connects, then we pass the new client\_sock to a handler method.
* **Starting the Server:** You can start this thread at application launch or when the user clicks a "Start Session" button. For example:
* server\_thread = DeviceServerThread(host='0.0.0.0', port=9000)  
  server\_thread.start()
* This will begin listening in the background.
* **IDE/Environment Setup:** If using an IDE (like PyCharm, VSCode, etc.), no special configuration is required to run threads. Just run the main GUI script; the server thread will start and run in parallel. Ensure that if you run the app multiple times, the previous instance closed properly and freed the port. In PyCharm, you can set up a Run/Debug configuration for the main script. While debugging, be cautious that breakpoints in the server thread will pause that thread and potentially block new connections; for testing, you might rely on print/log outputs instead of frequent breakpoints in the accept loop.
* **Firewall Consideration:** On first run, your OS might prompt to allow the Python app to listen on the network (especially on Windows). Allow access so that external devices (Android phones) can connect. If needed, manually add a firewall rule for port 9000 or run the app as administrator to bind to the port if required.

At this point, the server should be up and listening. Next, we will handle incoming connections and device registration.

### 2. Accepting Connections and Device Registration

**Goal:** When a new device connects to the server, establish the connection and register the device in the application. The Android device is expected to send an initial **hello message** (in JSON format) upon connecting, containing its unique ID and capabilities. The PC app should parse this message and create a representation of the device in the system.

**Implementation Steps:**

* **Spawn a Device Handler Thread:** Inside DeviceServerThread.handle\_new\_client(), create a new thread dedicated to this client. For example:
* import json  
  class DeviceHandlerThread(threading.Thread):  
   def \_\_init\_\_(self, sock, addr, manager):  
   super().\_\_init\_\_(daemon=True)  
   self.sock = sock  
   self.addr = addr  
   self.manager = manager # reference to a DeviceManager or main UI to callback  
   self.device\_id = None # will be set after receiving hello  
   self.running = True  
    
   def run(self):  
   try:  
   # Immediately expect a hello message from device  
   hello\_data = self.\_recv\_json() # method to receive one JSON message  
   if hello\_data and hello\_data.get("type") == "hello":  
   self.device\_id = hello\_data.get("id", "<unknown>")  
   caps = hello\_data.get("capabilities", {})  
   # Register device with the manager  
   self.manager.register\_device(self.device\_id, caps, self)  
   # Now enter main loop to receive further messages  
   self.listen\_loop()  
   else:  
   print("Did not receive hello from", self.addr)  
   self.sock.close()  
   except Exception as e:  
   print(f"Error in device handler {self.addr}: {e}")  
   self.sock.close()  
    
   def listen\_loop(self):  
   # loop receiving messages until disconnected  
   while self.running:  
   data = self.\_recv\_json()  
   if not data:  
   break # connection closed  
   self.handle\_message(data)  
   # If we exit loop:  
   self.manager.handle\_disconnect(self.device\_id)  
   self.sock.close()  
   print(f"Connection to {self.device\_id} closed.")  
    
   def \_recv\_json(self):  
   # Helper to receive one JSON object (assuming newline-delimited messages)  
   buffer = b""  
   while b"\n" not in buffer:  
   chunk = self.sock.recv(4096)  
   if not chunk:  
   return None # connection closed  
   buffer += chunk  
   # Split at newline (supports multiple messages in buffer)  
   line, \_, rest = buffer.partition(b"\n")  
   # If there's extra data after one message, keep it for next time (you could store it)  
   buffer = rest  
   # Parse JSON  
   try:  
   return json.loads(line.decode('utf-8'))  
   except json.JSONDecodeError:  
   print("Received invalid JSON from", self.addr)  
   return None  
    
   def handle\_message(self, data):  
   # Will implement in next steps for different message types  
   pass

In this snippet: - We expect the **hello** message right after connection. The \_recv\_json method reads from the socket until it finds a newline \n, then decodes one JSON message. - If the message is of type "hello", we extract a device identifier and capabilities. The device may send an ID ("id") or we might use the socket address if no ID provided. - We then call manager.register\_device(), which is a method to create a RemoteDevice instance and update the UI (we'll define this soon). - After registration, we enter a listen\_loop() to continually read messages until the connection is closed or an error occurs. Each message is passed to handle\_message() for processing. - If \_recv\_json() returns None, that indicates the socket was closed (no data). We break out and handle disconnection.

* **Device Manager and Device Registration:** We should have a central manager (maybe the main window or a dedicated class DeviceManager) to keep track of devices. For example:
* class DeviceManager:  
   def \_\_init\_\_(self, ui):  
   self.ui = ui # reference to UI/main window to update interface  
   self.devices = {} # dict of device\_id -> RemoteDevice  
   # Signals could be defined here if using PyQt signals for thread-safe updates.  
    
   def register\_device(self, device\_id, capabilities, handler\_thread):  
   # Create a RemoteDevice object to store info  
   device = RemoteDevice(device\_id, capabilities, handler\_thread)  
   self.devices[device\_id] = device  
   # Update UI (e.g., add device to list widget or table)  
   self.ui.add\_device\_to\_list(device)  
   print(f"Registered new device: {device\_id} with caps: {capabilities}")  
    
   def handle\_disconnect(self, device\_id):  
   if device\_id in self.devices:  
   self.devices[device\_id].is\_connected = False  
   # Update UI to mark device offline  
   self.ui.mark\_device\_offline(device\_id)  
   print(f"Device {device\_id} disconnected.")

And define a simple data class for RemoteDevice:

class RemoteDevice:  
 def \_\_init\_\_(self, device\_id, capabilities, handler\_thread):  
 self.id = device\_id  
 self.capabilities = capabilities # e.g. {"camera": True, "imu": False, ...}  
 self.handler = handler\_thread # reference to its thread (for sending data)  
 self.is\_connected = True  
 self.last\_seen = datetime.now()  
 # Additional fields: battery status, etc., can be added as needed.

The DeviceManager.register\_device method is called from the device thread when a "hello" is received. It creates a device entry and updates the UI. The UI update might involve adding a row to a QTableWidget or QListWidget showing the device’s name/ID and capabilities. If the device sends a dictionary of capabilities, you might map that to icons or text in the UI (for example, show a camera icon if "camera": True, etc.).

* **UI Update (Device List Panel):** In the UI, you should have a section (maybe a list on the side or a status bar) that shows currently connected devices. When ui.add\_device\_to\_list(device) is called, implement it to add a new list entry. For example, if using a QListWidget:
* def add\_device\_to\_list(self, device):  
   # Create a display string or widget for device  
   text = f"Device {device.id} (Capabilities: {', '.join(device.capabilities)})"  
   self.deviceListWidget.addItem(text)  
   # Optionally, store a reference to device in the widget item for easy access
* If using a more complex UI element (like a QTreeWidget or custom widget per device), update accordingly. Also consider visual cues: e.g., green dot for connected.

At this stage, when a device connects and sends a hello, the PC app should log the connection, register the device, and the UI should reflect that "Device X is connected with capabilities Y". Next, we will implement handling of incoming messages beyond the initial handshake.

### 3. Handling Incoming Messages from Devices

**Goal:** Continuously receive messages from each connected device and take appropriate actions. Each device's handler thread will decode incoming JSON messages and then update the application state or UI based on the message content.

**Key message types and how to handle them:**

* **Preview Frames (type = "preview\_frame"):** The device sends a preview image frame (likely from its camera) encoded as a base64 string in JSON (e.g., {"type": "preview\_frame", "image": "<BASE64\_DATA>"}). The server should:
* Decode the base64 string to raw image bytes (base64.b64decode in Python).
* Convert the bytes to a QImage or QPixmap. For example:
* img\_bytes = base64.b64decode(data["image"])  
  pixmap = QtGui.QPixmap()  
  pixmap.loadFromData(img\_bytes) # let Qt infer image format (JPEG/PNG) from data
* or use QImage.fromData(img\_bytes) then QPixmap.fromImage(...). This yields a QPixmap which can be displayed in a QLabel.
* **Thread to UI update:** Since the decoding is done in the device thread (background thread), we must pass the QPixmap to the main GUI thread to actually update a QLabel (because GUI operations should happen on the main thread). We can do this via signals. For instance, define a signal in DeviceManager or MainWindow:
* # in DeviceManager or MainWindow (which is a QObject)  
  image\_received = QtCore.pyqtSignal(str, QtGui.QPixmap)
* Connect this signal to a slot that updates the corresponding QLabel:
* self.deviceManager.image\_received.connect(self.update\_device\_image)  
    
  def update\_device\_image(self, device\_id, pixmap):  
   # find the QLabel for this device (could store in a dict by device\_id)  
   label = self.previewLabels[device\_id]  
   label.setPixmap(pixmap)
* Now, from the device thread, we emit the signal with the new pixmap:
* self.manager.image\_received.emit(self.device\_id, pixmap)
* PyQt signals are thread-safe – emitting a signal from a background thread will safely invoke the connected slot in the GUI thread[[2]](https://stackoverflow.com/questions/68287979/pyqt5-are-pyqtsignals-thread-safe#:~:text=3). This ensures the UI update happens in the correct thread without direct manipulation from the worker thread.
* **Status Updates (e.g., type = "status\_update"):** The device might periodically send info like battery level, available storage, temperature, etc. For example: {"type": "status\_update", "battery": 85, "temp": 36.5}. When the handler receives such a message, update the RemoteDevice object and emit a signal or call a function to refresh the UI for that device's status display. For instance:
* # In handle\_message:  
  elif msg\_type == "status\_update":  
   device = self.manager.devices.get(self.device\_id)  
   if device:  
   # Update device attributes  
   if "battery" in data:  
   device.battery = data["battery"]  
   # ... other status fields  
   # Emit a signal to update UI display  
   self.manager.status\_updated.emit(self.device\_id, data)
* The UI can display battery percentage next to the device, so the slot for status\_updated would update a label or icon (e.g., change a battery icon's level).
* **Sensor Data (type = "sensor\_data"):** If devices stream sensor readings (like heart rate, accelerometer, etc.) and if we want to display them in real-time on the PC, handle similarly to status updates. Perhaps maintain a small live-updating chart or just show latest values. For now, we can log or print them to confirm receipt, or update a label if one exists (e.g., heart rate value label).
* **Acknowledgments (type = "ack" or errors):** These are responses from device to commands we sent. For example, after we send a "start\_record" command, the device might reply {"type":"ack", "command":"start\_record"} or {"type":"error", "command":"start\_record", "message":"Failed to start camera"}. The handler should capture these and inform the main app:
* On an "ack", maybe mark that device as recording (if the command was to start recording).
* On an "error", possibly display a warning in the UI (e.g., a QMessageBox or a status bar message). You can implement this by emitting a signal to the main thread with the ack/error details or by directly calling a method on the manager to handle it (since it's not directly a UI widget update but perhaps logging).
* **Notifications (type = "notification"):** For events like recording finishing, file saved, etc. The device might send a message like {"type":"notification", "event":"recording\_finished", "file":"data123.mp4"}. On receiving this, the PC could:
* Update device state (e.g., mark not recording, increment a counter of files, etc.).
* Possibly automatically initiate a file transfer from the device (if that’s a planned feature) or prompt the user that a new file is ready.
* For now, maybe just log it or show it in UI (e.g., append to a log view or label).

**Implementing DeviceHandlerThread.handle\_message:**

Inside the DeviceHandlerThread.handle\_message(self, data) method, implement logic like:

def handle\_message(self, data):  
 msg\_type = data.get("type")  
 if msg\_type == "preview\_frame":  
 img\_b64 = data.get("image")  
 if img\_b64:  
 try:  
 img\_bytes = base64.b64decode(img\_b64)  
 pixmap = QtGui.QPixmap()  
 if pixmap.loadFromData(img\_bytes):  
 # Emit signal to update UI  
 self.manager.image\_received.emit(self.device\_id, pixmap)  
 except Exception as e:  
 print(f"Error decoding image from {self.device\_id}: {e}")  
 elif msg\_type == "status\_update":  
 # Update stored status and UI  
 self.manager.update\_device\_status(self.device\_id, data)  
 elif msg\_type == "sensor\_data":  
 self.manager.update\_device\_sensor(self.device\_id, data)  
 elif msg\_type == "ack":  
 cmd = data.get("command")  
 print(f"Device {self.device\_id} acknowledged {cmd}")  
 self.manager.handle\_device\_ack(self.device\_id, cmd, data)  
 elif msg\_type == "error":  
 cmd = data.get("command")  
 err\_msg = data.get("message", "")  
 print(f"Device {self.device\_id} reported error on {cmd}: {err\_msg}")  
 self.manager.handle\_device\_error(self.device\_id, cmd, err\_msg)  
 elif msg\_type == "notification":  
 event = data.get("event")  
 print(f"Device {self.device\_id} notification: {event}")  
 self.manager.handle\_device\_notification(self.device\_id, event, data)  
 else:  
 print(f"Unknown message type from {self.device\_id}: {msg\_type}")

The DeviceManager (or main UI) would have methods like update\_device\_status, handle\_device\_ack, etc., which update the UI or internal state appropriately.

**Thread Safety Note:** All modifications to GUI elements must happen in the main thread. We use Qt signals (self.manager.image\_received.emit, etc.) to ensure thread-safe communication from worker threads to the UI. Qt’s signal-slot mechanism will handle invoking the connected slots in the GUI thread safely[[2]](https://stackoverflow.com/questions/68287979/pyqt5-are-pyqtsignals-thread-safe#:~:text=3). For updates that involve only internal data (like updating the RemoteDevice object’s properties), we can do that directly in the background thread as long as those data structures are designed to be thread-safe or are only touched by one thread at a time. To avoid race conditions (for example, two threads updating a shared data structure simultaneously), you might use threading Locks around critical sections if needed. However, in our case: - Each RemoteDevice is primarily handled by its own thread and perhaps occasionally by the main thread (when sending a command). We will ensure to manage that carefully. - The devices dictionary in DeviceManager might be accessed by multiple threads (e.g., adding a device in handler thread vs. iterating or sending commands in main thread). To be safe, you could use a threading.Lock to guard modifications (e.g., acquire before adding/removing devices). For two devices, the risk of contention is low, but it's good practice to think about it.

By now, we have a server that can accept devices, register them, and handle incoming messages. Next, we need to enable sending commands from the PC to the devices.

### 4. Sending Commands to Devices (Outgoing Messages)

**Goal:** Allow the PC application (typically via user actions in the UI) to send JSON-based commands to one or more connected devices. This could be triggered by buttons like "Start Recording", "Stop Recording", "Capture Photo", etc., in the UI. The Device Connection Manager should expose functions to send these commands through the open sockets.

**Implementation Steps:**

* **Command Format:** Define the JSON structure for commands. For example:
* Start recording: {"type": "start\_record", "filename": "test1", "duration": 60} (the device might interpret this as "record for 60 seconds to file test1").
* Stop recording: {"type": "stop\_record"}
* Other commands as needed (like switching camera mode, marker insertion, etc.).
* **Sending via Sockets:** Each DeviceHandlerThread holds its sock (socket object). To send a JSON message, we do:
* def send\_command(self, command\_dict):  
   try:  
   msg = json.dumps(command\_dict)  
   # Add newline delimiter to signal message boundary  
   self.sock.sendall(msg.encode('utf-8') + b'\n')  
   except Exception as e:  
   print(f"Failed to send to {self.device\_id}: {e}")  
   self.manager.handle\_disconnect(self.device\_id)
* Using sock.sendall ensures the entire message is sent; we append a newline to match our protocol of newline-delimited JSON.
* **DeviceManager Helper:** In the DeviceManager class, add methods to send commands easily:
* class DeviceManager:  
   # ... existing methods ...  
   def send\_command\_to\_device(self, device\_id, command\_dict):  
   device = self.devices.get(device\_id)  
   if device and device.is\_connected:  
   device.handler.send\_command(command\_dict)  
   else:  
   print(f"Device {device\_id} not available to send command.")  
    
   def send\_command\_to\_all(self, command\_dict):  
   for device\_id, device in self.devices.items():  
   if device.is\_connected:  
   device.handler.send\_command(command\_dict)
* These methods can be called by the UI event handlers. For example, if the user presses a "Start Recording on All Devices" button:
* def on\_start\_all\_clicked(self):  
   cmd = {"type": "start\_record", "filename": "session1", "timestamp": time.time()}  
   self.deviceManager.send\_command\_to\_all(cmd)
* Or if there's a UI element to start one device individually, call send\_command\_to\_device(device\_id, cmd).
* **Thread Safety for Sending:** Here, the main UI thread will be invoking sock.sendall potentially while the device thread is doing sock.recv. This is generally OK – one thread can read while another writes on the same socket. However, two threads **writing** to the same socket at the same time could intermix data. In our design, we funnel all outgoing commands for a given device through the DeviceManager/main thread (user actions), while the device thread mostly just reads. This minimizes simultaneous writes from multiple threads. If there was a scenario of multiple threads writing, we should protect socket writes with a lock. We can add a lock per device:
* class RemoteDevice:  
   def \_\_init\_\_(...):  
   ...  
   self.send\_lock = threading.Lock()
* And in send\_command, do:
* with self.send\_lock:  
   self.sock.sendall(...encoded json...)
* This ensures only one thread writes at a time for that socket. Given the low frequency of commands and that typically only the main thread writes, this is mostly precautionary.
* **Handling Command Responses:** As mentioned earlier, devices should send back an "ack" or "error" for commands. We have already set up handle\_message to catch those. For a smoother UX, you might implement a mechanism to wait for an ack (or timeout) when a command is sent, but that can complicate the flow. For now, simply sending the command and logging any response is sufficient. If needed, we could maintain a dictionary of "pending commands" with a callback or status that gets updated on ack.

At this point, the PC can instruct devices to perform actions. Next, let's ensure the system handles disconnections properly.

### 5. Handling Device Disconnections and Reconnections

**Goal:** Gracefully handle the scenario where a device disconnects (intentionally or due to error). The UI should update to reflect the device is offline, and the server should remain running to allow the device (or another) to reconnect.

**Possible disconnection scenarios:** - The user stops the device app or the device loses network – the socket will close. - We detect disconnection in our server when sock.recv() returns empty or throws an exception. - There could also be an explicit "goodbye" message (not specified, but possible).

**Implementation Steps:**

* **Detection:** In the DeviceHandlerThread.listen\_loop, if \_recv\_json() returns None (meaning no data, socket closed) or an exception is caught, we know the connection dropped. We already call self.manager.handle\_disconnect(self.device\_id) in that case (see previous code).
* **UI Update on Disconnect:** In DeviceManager.handle\_disconnect(device\_id), we should update the UI to indicate that device is offline. There are a few UI design possibilities:
* Remove the device from the list entirely.
* Keep it in the list but grey it out or add "(disconnected)" to its name.

For simplicity, we can remove it from the list:

def handle\_disconnect(self, device\_id):  
 device = self.devices.get(device\_id)  
 if device:  
 device.is\_connected = False  
 # Remove from UI list:  
 self.ui.remove\_device\_from\_list(device\_id)  
 # Optionally, keep it in devices dict or pop it out:  
 # self.devices.pop(device\_id, None)

Where remove\_device\_from\_list finds the item in the QListWidget (or whichever UI component) and removes it or updates its text/status.

* **Stopping Threads:** The device handler thread will naturally exit its loop when disconnection is detected. It should then terminate since run() finishes. Because we set threads as daemon, even if we forgot to join them, they won't block program exit. However, it’s good practice to ensure they terminate. If the socket closes, any blocking recv will unblock and return, causing our loop to break.
* **Allowing Reconnection:** Our server thread (in DeviceServerThread) is still running and listening for new connections. If the same device reconnects (perhaps after a crash or restart), it will likely send a hello again. We’ll create a new RemoteDevice instance or reuse the old entry. A simple approach is to remove the old one on disconnect (so devices dict is updated). If you want to preserve some history (like last known battery level), you might keep the object but mark disconnected. For now, we can remove it. If the device reconnects, it will appear as a fresh entry. Optionally, you can check if device\_id already existed and update that instead of creating a new one.
* **Concurrent Disconnections:** If one device disconnects while others remain, ensure we handle each independently. Our implementation calls handle\_disconnect from that device’s thread, which updates UI. No special handling needed in the server thread except continuing to accept new connections.
* **Server Shutdown:** If the user closes the PC application or stops the session, we should also shut down the server socket and threads:
* In the main server thread, set self.running = False and close the server socket. This will break out of the accept loop (the accept() will error, which we catch and break).
* Also instruct all device threads to stop (they will anyway when sockets close).
* One way to unblock accept() is to call server\_sock.close() or send a dummy connection from localhost to wake it up.
* Because we set threads as daemon, closing the app will forcefully stop them, but it’s cleaner to handle it gracefully.
* For now, note that if the app exits, threads will die since they are daemon. In a more robust design, you'd join threads on exit.

**Recap:** When a device disconnects, the UI is updated (e.g., device removed or marked offline), and any resources associated with it are cleaned up. The server remains active to accept new or returning connections. If the app is closed, ensure the server socket is closed and threads are terminated to avoid any lingering background processes.

### 6. Threading and Concurrency Considerations

**Multi-Threading Model:** We chose a simple threading model: one thread for accepting connections, and one thread per connected device for handling I/O. This is straightforward and sufficient for a small number of devices (the plan mentioned possibly 2 devices, which is easily handled with threads). Python’s Global Interpreter Lock (GIL) means only one thread executes Python bytecode at a time, but since socket I/O releases the GIL while waiting for data, multiple connections can be serviced without significant blocking. For a handful of devices, this overhead is minimal.

**Why not use async I/O or Qt Network classes?** While we could use asyncio or Qt’s QTcpServer/QAbstractSocket to handle multiple connections in a single thread, the complexity is higher. Threads provide a simpler mental model and given the device count is low, it’s an acceptable solution.

**GUI Thread Safety:** Emphasizing again that **GUI updates must happen in the main thread**. Any direct call from a background thread to a QWidget (like calling label.setText or label.setPixmap) can crash the application or cause unpredictable behavior. Instead, communicate via signals (as we implemented for images and status updates). Qt’s signals and slots are designed for exactly this purpose, and they are thread-safe for cross-thread communication[[2]](https://stackoverflow.com/questions/68287979/pyqt5-are-pyqtsignals-thread-safe#:~:text=3).

**Stopping Threads Gracefully:** - The server thread runs an accept() loop; we break it by closing the server socket. Marking it daemon ensures it won’t hang the app on close even if we forget to break. - The device threads run a loop on recv; they exit when the socket closes or an error occurs. We don’t usually need to force-stop them, but if needed we could set a flag self.running = False (like if we implement a “Disconnect” button on the PC side to drop a device). - If using QThread instead of threading.Thread, we could get more integration with Qt’s lifecycle (and have a quit() method). But for now, threading.Thread suffices.

**Locks and Shared Data:** - If multiple threads need to access/modify the same data (e.g., the central devices dict, or writing to a log file), use threading.Lock or other synchronization. In our design, we might add a lock for the devices dict when adding/removing, though operations are quick and low-contention. - Also consider a lock if two threads could send on the same socket (as discussed, typically only one thread will send for each socket). - The example from GeeksforGeeks used a lock around printing to avoid jumbled output when multiple threads print simultaneously[[3]](https://www.geeksforgeeks.org/python/socket-programming-multi-threading-python/#:~:text=c%2C%20addr%20%3D%20s,1%5D%29%20start_new_thread%28handle_client%2C%20%28c)[[4]](https://www.geeksforgeeks.org/python/socket-programming-multi-threading-python/#:~:text=,avoiding%20jumbled%20prints). In our context, if we print from multiple threads, prints could intermix. It's usually fine, but you could use a similar approach if debugging logs become confusing.

### 7. Class and Module Breakdown

To keep the project organized, we can separate the device connection logic into its own module and define clear classes for each component. Here’s a possible breakdown:

* **Module: network\_server.py** – contains classes related to networking:
* DeviceServerThread – the server listener thread (sets up socket, accepts connections).
* DeviceHandlerThread – handles communication with a single device (one instance per device).
* These classes don’t directly know about the UI; they might interact via callbacks or a manager.
* **Module: device\_manager.py** – contains:
* DeviceManager class – responsible for keeping track of devices, providing methods to send commands, and perhaps defining PyQt signals for communication with UI.
* RemoteDevice class – data structure for device info/state.
* **Main UI Module (e.g., main\_window.py)** – the PyQt main window:
* Holds an instance of DeviceManager.
* Starts the DeviceServerThread (perhaps via DeviceManager.start\_server() which in turn creates the thread).
* Connects signals from DeviceManager to UI update slots.
* Implements UI event handlers that call DeviceManager.send\_command... methods.

This separation allows for easier maintenance. For instance, the networking code can be unit-tested or run in isolation (with dummy data) without needing the full UI. It also makes it clearer where each piece of functionality resides.

**Class Responsibilities:**

* DeviceServerThread:
* Listens on a socket for new connections.
* On accept, logs the connection and starts a DeviceHandlerThread for it.
* Should handle shutdown of the listening socket when needed.
* DeviceHandlerThread:
* Manages the socket communication with one device.
* Reads incoming data, parses JSON messages.
* On first message (hello), identifies and registers the device via DeviceManager.
* Thereafter, processes each message (calls appropriate handler in DeviceManager or emits signals).
* Provides a method to send commands (send\_command) to the device.
* On socket closure or error, informs DeviceManager about disconnection.
* DeviceManager:
* Stores a dictionary of active devices (RemoteDevice instances).
* Provides methods to register new devices and handle their removal.
* Defines Qt signals (if using PyQt) to communicate with UI for events like new device, device removed, image received, status update, etc.
* Provides methods for sending commands to devices (utilizing DeviceHandlerThread.send\_command).
* Possibly handles higher-level logic, e.g., broadcasting a command to all devices, or aggregating data if needed.
* Could also manage logging of events if desired.
* RemoteDevice:
* Represents the state of a device (ID, capabilities, connection status, last known statuses such as battery, plus a reference to its handler thread/socket).
* This is mostly a data container, possibly with some helper methods if needed (for example, a method to nicely format its name or capabilities).

**IDE Project Configuration:**

If using an IDE, ensure that all these modules are part of the project. You might structure the project as:

/project\_folder  
 main.py (or main\_window.py)  
 network\_server.py  
 device\_manager.py  
 ui/ (maybe .ui files or icons)  
 ...

In PyCharm, mark the root as the Sources Root if needed so imports like from device\_manager import DeviceManager work. Similarly, if using Visual Studio Code, just open the project folder and it should work. No special configurations beyond the normal Python interpreter and PyQt installation are needed.

Ensure PyQt5 or PySide6 (whichever you use) is installed in your environment. If not, install via pip (e.g., pip install PyQt5).

### 8. Testing and Verification Plan

With the implementation in place, thorough testing is crucial. We will test the Device Connection Manager in incremental steps:

**Step 1: Basic Server Startup**  
- **Test:** Run the PC application and ensure that the server thread starts without errors and listens on port 9000.  
- **Verification:** You should see in the console (or log) the message "Server listening on 0.0.0.0:9000". No UI changes yet, just confirming no exceptions on startup. Optionally, use a tool like netstat to verify the port is open, or try connecting via a network tool.

**Step 2: Simulated Device Connection (Hello Message)**  
- **Test:** Simulate a device connecting and sending a hello message. If you don't have the Android app ready, use a Python script or telnet. For example, run this in a separate Python interpreter:

import socket, json  
s = socket.socket()  
s.connect(("127.0.0.1", 9000)) # connect to server (use PC's IP if running externally)  
hello\_msg = {"type": "hello", "id": "DeviceA", "capabilities": {"camera": True, "imu": True}}  
s.sendall((json.dumps(hello\_msg) + "\n").encode('utf-8'))

- Alternatively, use a tool like **netcat**:

nc 127.0.0.1 9000

and then paste the JSON string followed by Enter. - **Verification:** The PC app should log "Accepted connection..." and "Registered new device: DeviceA..." in the console. In the UI, you should see a new device entry (e.g., "DeviceA (Capabilities: camera, imu)") appear in the device list panel. This confirms that the accept loop, device thread, JSON parsing, and UI update for new device are working.

**Step 3: Receive a Preview Frame**  
- **Test:** Simulate a preview\_frame message from the device. Continuing from the previous simulation, after sending the hello (and not closing the socket), send a fake preview frame:

import base64  
# Create a dummy small image data, e.g., 1x1 pixel PNG for simplicity  
dummy\_image\_bytes = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x01...<rest of PNG data>'  
# (For brevity, you could prepare a base64 string of a valid small image. Or skip actual image content.)  
b64\_image = base64.b64encode(dummy\_image\_bytes).decode('utf-8')  
frame\_msg = {"type": "preview\_frame", "image": b64\_image}  
s.sendall((json.dumps(frame\_msg) + "\n").encode('utf-8'))

(Ensure the dummy image bytes form a valid image or the QPixmap.loadFromData might fail. Alternatively, test with an actual image file read into bytes and base64 encoded.) - **Verification:** The PC app’s device thread should decode the message and emit the signal. In the UI, the preview QLabel for "DeviceA" should get updated with the image (if everything is correct and an image was provided). If using a dummy image that’s not valid, at least observe that no crashes occur and maybe log an error about decoding. This step confirms the image path (receive -> decode -> signal -> UI) is working.

**Step 4: Receive a Status Update**  
- **Test:** Simulate a status update message:

status\_msg = {"type": "status\_update", "battery": 78, "temp": 37.2}  
s.sendall((json.dumps(status\_msg) + "\n").encode('utf-8'))

- **Verification:** Check the PC app console for any print like "update\_device\_status" and ensure no error. In the UI, if you have a battery indicator for the device, it should update (e.g., show 78%). If not explicitly visible, you can add a temporary log or label to display battery for testing. This verifies that JSON parsing and updating device info works.

**Step 5: Acknowledgment handling**  
- **Test:** Simulate an ack to a command: - First, have the PC send a command (see next step) and then simulate device ack. - Or simply simulate an ack out of the blue:

ack\_msg = {"type": "ack", "command": "start\_record"}  
s.sendall((json.dumps(ack\_msg) + "\n").encode('utf-8'))

- **Verification:** The PC should print that DeviceA acknowledged start\_record. If the UI had any indicator (like a recording status icon), it could change state now. This confirms the ack path works.

**Step 6: Sending a Command from PC to Device**  
- **Test:** In the PC app UI, perform the action that sends a command. For example, click the "Start Recording" button which triggers DeviceManager.send\_command\_to\_device("DeviceA", {"type":"start\_record",...}). Ensure the dummy client prints or logs what it receives:

data = s.recv(1024)  
print("Received on device:", data)

Or if using netcat, you'll see the JSON printed in that console. - **Verification:** The dummy client should receive the JSON command that was sent. This verifies that the PC properly serialized and sent the command. Also confirm the PC didn't throw any errors on sending. If the dummy client echoes an ack back (you can manually send it as above), the whole request-response cycle is tested.

**Step 7: Device Disconnect**  
- **Test:** Simulate the device disconnecting. In the dummy script, simply close the socket:

s.close()

If using netcat, just Ctrl+C to terminate it. - **Verification:** The PC app should detect the disconnection. In the console, it may show an error or the handle\_disconnect print. The UI should update (DeviceA removed or marked offline). Ensure no crashes occur on disconnect. Also, verify the server thread is still running (you can attempt to connect again). Optionally, try reconnecting the dummy client:

s2 = socket.socket()  
s2.connect(("127.0.0.1", 9000))  
s2.sendall((json.dumps(hello\_msg) + "\n").encode())

It should register as DeviceA again (or DeviceA(2) if you choose to differentiate). This ensures reconnections are handled.

Throughout testing, keep an eye on the **application logs/console** for any exceptions or error messages. Any unhandled exceptions in threads might not crash the whole app but could indicate logic issues that need fixing (for instance, JSON decode errors, KeyError on missing fields, etc.).

**Test with Multiple Devices:** If possible, simulate two devices connecting concurrently: - Start two dummy client instances (or have one script open two sockets) and send hellos for "DeviceA" and "DeviceB". - Both should appear in the UI. - Send a preview frame from each (maybe with different dummy images), ensure each goes to the correct UI element. - Send a command from PC to all, verify both receive it. - Disconnect one, verify the other remains unaffected.

This will test the multi-threaded handling and ensure no cross-talk or resource conflicts.

### 9. Additional Tips and Considerations

* **Android Client Implementation:** Ensure the Android app is configured to connect to the PC’s IP address on port 9000. Usually, when phone and PC are on the same Wi-Fi network, you can use the PC’s local IP. If testing with an Android emulator, remember that 10.0.2.2 is the special IP to reach the host PC. These details will be handled in the Android app, but double-check when integrating.
* **JSON Message Framing:** We chose a simple newline \n delimiter for JSON messages. This is straightforward but requires that the JSON text does not contain newlines itself (or if it does, the device should escape or remove them). An alternative is to send a length prefix (e.g., send 4 bytes of length followed by that many bytes of JSON) to know how many bytes to read. Our \_recv\_json method accumulates bytes until a newline is seen, which should work fine as long as the device app sends each JSON message followed by a newline (this is something to implement on the Android side). If you run into issues with messages splitting or concatenation, consider refining the protocol (e.g., use \n and ensure device sends one JSON per line, or use a special delimiter like \0).
* **Error Handling and Robustness:** In production, you’d want more robust error handling:
* If a JSON message is malformed, decide whether to drop it or close connection. Currently, we log and ignore.
* If a device sends an unexpected type, we handle it in the default branch.
* If the PC fails to send (maybe device disconnected right when sending), we catch the exception and handle disconnect.
* Timeouts: sockets by default may block indefinitely on recv. You could set a socket timeout (e.g., 5 seconds) and if no data in a while, send a ping or consider it dropped. Alternatively, use a separate heartbeat message. For simplicity, we didn't implement a heartbeat, but it's something to consider if devices might silently drop.
* **Integration with UI/UX:** Once the backend is working, integrate with the actual UI flows:
* Maybe disable the "Start Recording" button until at least one device is connected.
* Maybe show a status bar message like "2 devices connected" and update as they come/go.
* Ensure that when the app closes, it stops the server thread (to avoid "Address already in use" on next launch if the socket remains open for a while).
* **Logging:** During development, printing to console is fine for observing behavior. In a real application, you might use Python’s logging module to log info/warnings to a file, which can help in debugging issues in field usage.
* **Extensibility:** The Device Connection Manager now provides a backbone for communication. Future milestones (likely involving starting recordings, collecting data files, etc.) will build on this:
* e.g., when a recording finishes (device sends notification), the PC might automatically download the file from the device (maybe via a separate socket or an HTTP endpoint, depending on design).
* You might also incorporate a way to send a “stop session” command to all devices and then shut down the server.

By following this guide and testing each part, you should have a fully functional network communication layer between the PC and the Android devices. This will serve as the foundation for the multi-sensor recording system’s coordinated control and data collection.

[[1]](https://www.geeksforgeeks.org/python/socket-programming-multi-threading-python/#:~:text=host%20%3D%20%27%27%20port%20%3D,port) [[3]](https://www.geeksforgeeks.org/python/socket-programming-multi-threading-python/#:~:text=c%2C%20addr%20%3D%20s,1%5D%29%20start_new_thread%28handle_client%2C%20%28c) [[4]](https://www.geeksforgeeks.org/python/socket-programming-multi-threading-python/#:~:text=,avoiding%20jumbled%20prints) Socket Programming with Multi-threading in Python - GeeksforGeeks

<https://www.geeksforgeeks.org/python/socket-programming-multi-threading-python/>

[[2]](https://stackoverflow.com/questions/68287979/pyqt5-are-pyqtsignals-thread-safe#:~:text=3) python 3.x - PyQt5 : are pyqtSignals thread safe? - Stack Overflow

<https://stackoverflow.com/questions/68287979/pyqt5-are-pyqtsignals-thread-safe>