Chapter 1

Chapter 4: Design and Implementation

1.1 4.1 System Architecture Overview (PC—Android System Design)

The **Multi-Sensor Recording System** is implemented as a distributed architecture consisting of an Android mobile application and a Python-based desktop controller. The Android device functions as a sensor node responsible for data capture, while the PC acts as a central coordinator or hub. This **PC—Android system design** follows a master—slave paradigm in which the Python desktop application orchestrates one or more Android sensor nodes, achieving precise synchronized operation across all devices [**?**] [**?**]. The design balances device autonomy with centralized control: each Android device can operate independently for local sensor management and data logging, yet all devices participate in coordinated sessions managed by the desktop controller for unified timing and control.

The architecture emphasizes **temporal synchronization, reliability, and modularity**. A custom network communication layer links the mobile and desktop components, enabling command-and-control messages, status updates, and data previews over a Wi-Fi or LAN connection. The system employs event-driven communication patterns with robust error handling and recovery, ensuring that transient network issues or device glitches do not compromise the entire session [**?**]. Each device buffers and locally stores data so that even if connectivity is lost momentarily, data collection can continue uninterrupted; once the connection is restored, the system can realign the data streams in time [**?**]. This fault-tolerant approach, combined with comprehensive logging on both mobile and PC sides, guarantees data integrity and consistency throughout a recording session.

Figure 4.1: System architecture overview of the multi-sensor recording system. This diagram depicts the central desktop controller (PC) communicating with one or more Android devices over a network. Each Android device interfaces with onboard and external sensors (cameras, thermal sensor, GSR sensor) and handles local data acquisition and storage. The desktop controller provides a GUI for the user and runs coordination services (network server, synchronization engine, data manager), sending control commands to the Android app and receiving live status and preview data. The design shows a hybrid star topology: the PC is the hub coordinating distributed mobile nodes, enabling synchronized start/stop triggers, real-time monitoring, and

unified timekeeping across the system.

1.2 4.2 Android Application Design and Sensor Integration

The **Android application** is a comprehensive sensor data collection platform that integrates the phone's native sensors (e.g. camera) with external devices. It is developed in Kotlin and structured using a clear **layered architecture** to separate concerns. The app follows an MVVM (Model—View—ViewModel) design, where a thin UI layer (Activities/Fragments and ViewModels) interacts with a robust **business logic layer** of managers and controllers, which in turn utilize lower-level sensor interfacing components. This design maximizes modularity and maintainability, allowing each sensor modality to be managed independently while ensuring all subsystems remain synchronized. Key architectural components include a SessionManager class for coordinating recording sessions, a `DeviceManager` for handling attached sensor devices, and a `ConnectionManager` for managing the network link to the PC controller [**?**]. The data acquisition layer comprises specialized recorder classes for each modality — e.g. a CameraRecorder class for the phone's RGB camera, a ThermalRecorder class for the USB thermal camera, and a ShimmerRecorder class for the GSR sensor — each encapsulating the details of interfacing with that sensor hardware [**?**]. These recorders feed sensor data into the session management framework and ultimately into local storage, while also forwarding preview data through the network layer for remote monitoring. The app makes heavy use of Android's asynchronous capabilities (threads, Handler, and coroutines) to handle high data rates and multiple sensors in parallel, ensuring that operations like writing to storage or processing sensor inputs do not block the user interface. Dependency injection (via Hilt) is utilized to manage the complexity of cross-cutting concerns like logging and configuration, further decoupling components.

Importantly, the Android application is built to facilitate **precise time alignment** of multimodal data at the point of capture. All sensor readings and frames are timestamped using a common reference (system clock or a synchronized clock source) as they are recorded. For example, when a recording session begins under remote command, the app initializes each sensor nearly simultaneously and tags the data with timestamps that can later be correlated across devices. The Android app also includes on-device preprocessing features — for instance, a **hand region segmentation** module uses MediaPipe to detect hand landmarks in the camera frame in real-time [**?**]. This can aid in focusing analysis on specific regions of interest (such as the subject's hand or face in the thermal imagery) without requiring external post-processing. The overall Android design thus serves as a flexible yet controlled data collection node that seamlessly integrates heterogeneous sensors under a unified workflow.

(Figure 4.2: Layered design of the Android application. The figure outlines the app's architecture with four layers: a presentation layer (UI and ViewModels for user interaction and state), a business logic layer (managers like SessionManager, DeviceManager, and ConnectionManager coordinating recording and devices), a data acquisition layer (sensor-specific recorder modules for camera, thermal, and GSR, as well as processing components like hand segmentation), and an infrastructure layer (network communication client, local storage handlers, and performance monitors). Arrows illustrate the flow: user actions in the UI invoke ViewModel updates, which

delegate to managers that control the sensor recorders. The recorders produce data that is stored locally and also sent through the network layer to the PC. This diagram highlights modular separation: each sensor integration is implemented in its own module, all orchestrated by the central session manager.)

1.2.1 4.2.1 Thermal Camera Integration (Topdon)

One of the distinguishing features of the system is its **thermal imaging capability**, achieved by integrating a Topdon TC001 thermal camera with the Android device. The Topdon thermal camera is a USB-C accessory providing infrared imaging, and it is supported in the app via the manufacturer's SDK. The integration was designed to enable **real-time thermal video capture** alongside the phone's regular camera. To use the thermal camera, the Android device serves as a USB host (via OTG), and the app interfaces with the camera through the SDK's APIs for device discovery, configuration, and frame retrieval [**?**]. When the thermal camera is connected, the app's `ThermalRecorder` component handles the entire lifecycle: it listens for USB attach events, requests permission from the Android USB system to access the device, and initializes the camera feed [**?**] [**?**]. The Topdon TC001 supports a sensor resolution of 256×192 pixels with a frame rate of 25 FPS, which the app configures as the default thermal video mode [**?**]. These parameters (resolution, frame rate, calibration settings) can be adjusted via a `ThermalCameraSettings` configuration to trade off image detail vs. performance, but by default the system uses the full available resolution and a frame rate that matches typical thermal camera capabilities.

Captured thermal frames consist of both a thermal image (usually represented as a color or grayscale thermogram) and underlying temperature data for each pixel. The `ThermalRecorder` obtains each frame from the SDK callback in a background thread to avoid stalling the UI. Each frame is timestamped with a high-resolution timestamp (synchronized to the system clock or a master clock) and placed into a queue for processing and storage. The app writes the raw thermal data stream to a file in real-time during recording — typically this is a proprietary binary format that includes a header (with metadata like resolution, frame rate, and possibly calibration parameters) followed by a sequence of frames, each prefixed with its timestamp [**?**] [**?**]. This raw log allows precise post-hoc analysis of temperature values. In parallel, the app can generate visible thermal images (e.g., JPEG frames) from the raw data for quick preview or user feedback. The `ThermalRecorder` optionally provides a downsampled live preview feed: it converts incoming thermal frames to a viewable image (applying a colormap and scaling) and streams these preview frames over the network to the desktop controller [**?**]. This gives the researcher immediate visual feedback of the thermal camera's view, which is invaluable for ensuring the sensor is aimed correctly and functioning during a session.

Integrating the Topdon camera posed several challenges which were addressed in design. First, USB power and bandwidth management on the mobile device had to be considered — the app monitors device attach/detach events and gracefully handles unexpected disconnects (for example, if the camera is unplugged mid-session, the system logs a warning and the recorder stops, but other sensors continue unaffected). Additionally, to maintain synchronization with other data, the thermal frames are timestamped in the same epoch as the phone's video frames

and GSR samples; this enables the **thermal data to be temporally aligned** with the RGB video and physiological signals during analysis. The result is a tightly coupled thermal imaging module that extends the Android phone's sensing capabilities with minimal latency. **Thermal camera integration is fully incorporated into the session workflow** — the user can toggle thermal recording on or off for a session, and if enabled, the system will automatically initialize the Topdon device and begin capturing when the session starts (under PC command), then finalize and close the device when the session ends. All these steps occur behind the scenes, preserving a seamless user experience.

(Figure 4.3: Thermal camera integration flow. This figure illustrates how the Android app interfaces with the Topdon TC001 thermal camera via USB. When the camera is plugged in, the app's USB listener requests permission from the Android OS and initializes the Topdon SDK. During a recording session, the app continuously pulls thermal frames from the camera at ˜25 Hz. Each frame is time-stamped and written to local storage as part of a thermal data file, and simultaneously a scaled preview image is sent over the network to the PC for real-time monitoring. The diagram also highlights the coordination required: the PC's "start recording" command triggers the camera initialization (opening the USB device and starting capture) almost concurrently with other sensors, ensuring the thermal stream is synchronized with the overall session timeline.)

1.2.2 4.2.2 GSR Sensor Integration (Shimmer)

In addition to imaging, the system incorporates a **physiological sensor** — specifically, a Shimmer3 GSR+ device — to record galvanic skin response (electrodermal activity), and optionally other signals like photoplethysmogram (PPG) and motion from the Shimmer's on-board sensors. The Shimmer3 GSR+ is a research-grade wearable sensor that connects via Bluetooth. The Android application's ShimmerRecorder class module manages the **Bluetooth communication** and data logging for this device. On application startup or when the user chooses to connect a Shimmer sensor, the app scans for available Bluetooth devices and pairs with the Shimmer (using its default PIN and name if needed). The integration leverages the Shimmer Android SDK provided by Shimmer Research, which offers an API (`ShimmerBluetoothManagerAndroid` and related classes) to handle the low-level Bluetooth link and streaming of sensor data [**?**] [**?**]. Once connected, the app configures the Shimmer device to enable the desired sensor channels and sampling rate. By default, the system activates the GSR channel (skin conductance) on the Shimmer, as well as the PPG channel and basic kinematic channels (accelerometer axes) for context, using a sampling rate of 51.2 Hz for physiological signals [**?**]. The GSR range is set to an appropriate setting (e.g. ±4 µS range) to capture typical skin conductance levels [**?**]. These defaults can be adjusted through the app's settings interface if needed.

Data from the Shimmer sensor arrives as a stream of packets, each containing a set of measurements (GSR, PPG, etc.) with a timestamp from the Shimmer's internal clock. The ShimmerRecorder class runs a dedicated handler thread that listens for incoming data packets via the Shimmer SDK's callback system [**?**] [**?**]. As each packet is received, the app immediately records a corresponding system timestamp and then parses the sensor values. The data is buffered and written to a CSV text file in real-time, which serves as the session log for physiological data.

A single line in this CSV file might include: a timestamp (in milliseconds) from the phone's perspective, the original device timestamp from the Shimmer (for reference or redundancy), the GSR conductance value (in microsiemens), PPG readings (raw or processed), acceleration values, and the sensor's battery level [?]. For example, the header used in the CSV clearly enumerates these fields: *Timestamp_ms, DeviceTime_ms, SystemTime_ms, GSR_Conductance_uS, PPG_A13, Accel_X_g, ... Battery_Percentage*, etc. [?]. By storing both the device-reported time and the system time on reception, the system can later assess any clock drift or transmission delay and correct for it in analysis.

Similar to the thermal module, the Shimmer integration includes a **live data streaming** capability for monitoring. The ShimmerRecorder class can forward sampled data (e.g., the latest GSR value) over the network socket to the desktop in real-time (the code maintains an optional socket connection for sensor streaming on a specified port) [?] [?]. On the desktop side, these incoming samples could be displayed as a live graph or used to trigger alerts if, say, a physiological threshold is exceeded during the experiment. However, the primary storage of GSR data is on the Android device, ensuring no data loss if the network is lagging. The system also has built-in safeguards: if the Bluetooth connection to the Shimmer drops during recording, the app will attempt to reconnect up to a few times (with short delays) automatically [?]. All reconnection attempts and any data gaps are logged. In practice, maintaining a stable Bluetooth link was a known challenge due to interference and mobile OS power management, so the implementation uses Android's modern Bluetooth APIs (with runtime permission handling for Android 12+ where coarse and fine location permissions are required to scan/connect) [?] [?]. By handling both the legacy and new permission models, the app ensures compatibility across a range of Android OS versions.

Overall, the **Shimmer GSR integration** extends the Android app with the capability to capture high-quality physiological signals in sync with video and thermal data. The modular design of the ShimmerRecorder class means this component can start and stop recording in tandem with other sensors under the control of the central session manager. When a session begins, the app (upon receiving the command from PC) will initialize the Bluetooth link, start the data stream, and begin logging GSR. When the session ends, it closes the connection and finalizes the CSV file. The data recorded provides a ground-truth physiological timeline (skin conductance changes over time) that can be later correlated with the subject's visual and thermal data to draw insights about stress or arousal.

(Figure 4.4: GSR sensor (Shimmer3) integration. The figure shows the Shimmer GSR+ device wirelessly connected to the Android smartphone via Bluetooth. In a recording session, the Android app subscribes to the Shimmer's data stream, receiving packets that contain GSR and PPG readings. These data packets are timestamped and logged on the phone. The figure highlights the flow from sensor electrodes on the subject, through the Shimmer device's analog front-end (measuring skin conductance), transmitted over Bluetooth to the phone, and then into the app's data recording pipeline. Any loss of connection triggers the app's reconnection logic, ensuring continuity of data. A small real-time graph icon on the PC side suggests that as data is recorded, key values (like GSR level) are also sent to the desktop for live display.)

1.3 4.3 Desktop Controller Design and Functionality

The **desktop controller** is a Python application with a rich graphical user interface that serves as the command center for the entire multi-sensor system. It is built using the PyQt5 framework for the GUI, combined with a suite of backend services and managers that handle device communication, data management, and synchronization. Architecturally, the desktop application is divided into layers and components that mirror many responsibilities of the Android app, but at a higher coordination level. A **Presentation Layer** includes the main window and various UI panels (tabs) that the researcher interacts with: for example, a *Devices* tab to manage connected devices, a *Recording* tab to start/stop sessions and view live previews, a *Calibration* tab for camera calibration procedures, and a *Files/Analysis* tab for reviewing recorded data. This UI is designed to be intuitive, providing real-time feedback on system status (device connection state, battery levels, recording progress, etc.) [**?**].

Beneath the UI, the **Application Layer** of the desktop controller contains core logic components. The central piece is often called the `Application Controller` or `Session Manager` on the PC side — this orchestrates the overall workflow of a recording session (responding to user inputs from the UI, coordinating timing, and updating UI status). Complementing it are specialized managers such as a `DeviceManager` (to keep track of all connected Android devices and other sensors like USB webcams), a `CalibrationManager` (for handling multi-camera calibration routines using OpenCV), and possibly a `StimulusController` if the system supports presenting stimuli (like images or sounds) to the subject during the experiment [**?**]. Each of these components encapsulates a distinct piece of functionality, following a clear separation of concerns. For instance, the `DeviceManager` handles discovery of devices and maintaining connection info, but delegates the actual communication to a lower-level network service; the `CalibrationManager` encapsulates the procedures for capturing calibration images from cameras and computing calibration parameters without cluttering the main application logic.

At the next level, the desktop app includes a set of **Service Layer** or backend components that handle specific types of I/O and processing. Notable among these are the **Network Service**, which implements the socket server that listens for connections from Android devices, the **Webcam Service** for controlling any USB cameras attached to the PC (if the study uses external webcams in addition to phone cameras), the **Shimmer Service** for direct PC-to-Shimmer connectivity, and a **File Service** for managing data storage on the PC side [**?**]. The presence of both an Android Shimmer integration and a PC Shimmer service is intentional — the system is flexible to support different configurations. In scenarios where an Android phone is used by a subject, that phone might handle the Shimmer data as described in Section 4.2.2. However, the desktop application is also capable of directly connecting to Shimmer sensors via a Bluetooth dongle if needed (for example, in a lab setting where the PC is in range of the Shimmer, the researcher might choose to let the PC record GSR data directly). The design thus provides **multi-path integration** for sensors to improve robustness; the ShimmerManager class on the PC can accept data from either direct Bluetooth or through the Android (which relays it). This multi-library support with fallback ensures that even if one pipeline has an issue, the data can still be collected via the other [**?**].

All these services feed into the **Infrastructure Layer** on the PC, which includes cross-

cutting concerns like logging, synchronization, and error handling [?]. A dedicated **Synchronization Engine** runs on the desktop to maintain the master clock and align time across devices (details in Section 4.4). A global **Logging system** records events from all parts of the application (e.g., device connect/disconnect, commands sent, errors, etc.) for debugging and audit purposes [?]. An **Error Handler** and a **Performance Monitor** track the health of the system, issuing warnings or recovering from failures (for example, if a device disconnects unexpectedly, the UI will show an alert and the system will attempt reconnection or gracefully disable that device for the session) [?].

From a **functionality** perspective, the desktop controller provides the researcher with a one-stop interface to **manage multi-device recording sessions**. Using the UI, the user can configure an experiment session (select which devices/sensors are active, set participant or session metadata, etc.), then initiate a synchronized start. When the user hits S̈tart; the controller sends out start commands to all connected Android devices (and starts any local recordings like webcams or Shimmer) nearly simultaneously [?] [?]. During recording, the PC displays live previews — for instance, a thumbnail video feed from each Android's camera, as well as numeric readouts or simple plots of sensor data like GSR — giving confidence that all modalities are functioning. It also updates status indicators (battery levels of phones, available storage, current timestamp offsets, etc.) in real time. The PC periodically checks that all devices are still synchronized (drift monitoring) and can even warn if, say, an Android device's clock starts diverging or if data throughput from a device is lagging. When the user stops the session, the controller issues a synchronized stop command to all devices and awaits confirmation that each has safely finalized its data. It then collates metadata about the session (e.g., file names from each device, any timing offsets, calibration info) and can present a summary or save a session manifest.

Additionally, the desktop application includes utility features: a **camera calibration tool** (leveraging OpenCV — the researcher can collect images of a chessboard pattern from the different cameras to calculate calibration and alignment between, say, a phone's RGB camera and the thermal camera), and a **stimulus presentation module** which can display images or play audio on a connected screen as part of a study protocol. These are implemented as part of the UI and controlled through the same session manager to ensure any stimuli are timestamped and synchronized with the sensor data.

In summary, the desktop controller is the **brains of the system**, coordinating all pieces to work in unison. It abstracts the complexity of dealing with multiple devices by providing a unified interface and automating the low-level details of communication and timing. The use of Python with Qt and libraries like OpenCV, NumPy, and PySerial/Bluetooth gives it the power and flexibility needed for a research environment: it can be easily extended or scripted for new functionality (for example, adding support for another type of sensor or a new analysis routine) while maintaining real-time performance through optimized libraries and asynchronous design. The combination of a robust backend and an easy-to-use frontend makes the desktop application a critical component that bridges researchers with the distributed sensing network.

(Figure 4.5: High-level architecture of the Desktop Controller application. The figure breaks down the desktop software into its main components: a GUI layer (with windows/tabs for Record-

ing Control, Device Management, Calibration, etc.), an application logic layer (the main orchestrator and managers for sessions, devices, and calibration), and a service layer (which includes the network socket server, webcam interface, Shimmer interface, file and data management services). The diagram also shows an infrastructure layer beneath, containing the synchronization engine, logging system, and error handling modules that support the entire application. Arrows in the figure illustrate how user actions in the GUI propagate to the application layer (e.g., "Start Session" triggers the Session Manager), which then calls into various services (sending network commands, initializing webcams, etc.). Similarly, data flows upward: e.g., a preview frame from an Android device comes in through the Network Service and is passed to the GUI for display. The figure emphasizes modular design — each sensor or function has a dedicated service, coordinated by the central application logic, enabling easy maintenance and future scalability.)

1.4 4.4 Communication Protocol and Synchronization Mechanism

A core challenge of this project is enabling **reliable, low-latency communication** between the PC controller and the Android devices, along with a mechanism to synchronize their clocks for coordinated actions. The system addresses this with a custom-designed **communication protocol** built on standard networking protocols, and an integrated **synchronization service** that keeps all devices aligned to a master clock.

Communication Protocol: The Android app and desktop controller communicate over TCP/IP using a JSON-based messaging protocol. Upon startup, the desktop controller opens a server socket (by default on TCP port 9000) and waits for incoming connections [**?**]. Each Android device, when the app is launched, will initiate a connection to the PC's IP and port. A simple handshake is performed in which the Android sends an identifying message containing its device ID (a unique name or serial) and a list of its capabilities (e.g., indicating if it has a thermal camera, GSR sensor, etc.). The PC acknowledges and registers the device. All messages between PC and Android are formatted as JSON objects, with a length-prefixed framing (the first 4 bytes of each message indicate message size) to ensure the stream is parsed correctly. This design avoids reliance on newline or other delimiters that could be unreliable for binary data; instead, it robustly handles message boundaries, allowing binary payloads (like images) to be transmitted if needed by encoding them (often images are base64-encoded within JSON). The protocol defines several message types, including: *control commands* (e.g., `start_recording`, `stop_recording`), *status updates* (e.g., the Android periodically sends a `status` message with battery level, free storage, current recording status, etc.), *sensor data messages* for streaming (such as `preview_frame` for a JPEG preview image, or `gsr_sample` for a live GSR data point), and *acknowledgments* (the devices reply to key commands with an ACK/NACK to confirm receipt). The PC's network service is multi-threaded and non-blocking — it can handle multiple device connections simultaneously and route messages to the appropriate handlers, emitting Qt signals that update the UI or trigger internal logic. The communication is two-way: the PC can send commands to all or individual devices, and devices send asynchronous updates or data back to the PC.

To support different data exchange needs, the system effectively implements multiple logical channels over this link. For example, high-frequency binary data like video preview frames or sensor streams are sent in a compact form (with minimal JSON overhead aside from a message header), whereas less frequent control commands use a more verbose but human-readable JSON structure. In conceptual terms, we can think of a **control channel** and a **data channel** operating over the same socket. In future or in extensions, the design also allots a separate **file transfer mechanism** (e.g., an offline file download after recording, or an HTTP/FTP transfer) if large recorded files on the phone need to be pulled to the PC; however, in the current implementation, file transfer is often done manually after sessions or through external means, and the focus of the communication protocol is on real-time coordination and monitoring. All communications happen over the local network (typically the devices are on the same Wi-Fi or Ethernet LAN). Security is not heavily emphasized in this research prototype (messages are unencrypted JSON), but the system can be isolated on a private network during experiments for safety.

Synchronization Mechanism: Achieving **time synchronization** across devices is critical because we want, for instance, a thermal frame and a GSR sample that occur at the "same time" to truly represent the same moment. In a distributed system with independent clocks, our approach is to designate the desktop PC as the **master clock** and synchronize all other devices to it. The desktop controller runs a component called the `MasterClockSynchronizer` (or Synchronization Engine) which fulfills two primary roles: it distributes the current master time to clients (devices) and coordinates simultaneous actions based on that time. Concretely, the PC launches a lightweight **NTP (Network Time Protocol) server** on a UDP port (default 8889) to which devices can query for time. The Android app, upon connecting, performs an initial clock sync handshake — this can be a custom sync message or an NTP query — to measure the offset between its local clock and the PC clock. Given the typical latencies on a local network are low (on the order of a few milliseconds), this offset can be estimated with high precision using techniques akin to Cristian's algorithm or NTP's exchange (the system may send a timestamped sync message and get a response to calculate round-trip delay and clock offset). The `SynchronizationEngine` on the PC possibly refines this by periodic pings (e.g., every 5 seconds) to adjust for any drift during a long session. In practice, the Android device will apply any calculated offset to its own timestamps for data labeling, meaning if its clock was 5 ms ahead of the PC, it will subtract 5 ms from all timestamps to align with the master timeline.

When a recording session is initiated, the PC doesn't just send a blind "start" command — it issues a **coordinated start time**. For example, the PC might determine "start recording at time T = 1622541600.000 (Unix epoch seconds)" a few hundred milliseconds in the future, and send a message to each device: *"start_recording at T with session_id X"*. Each Android device receives this and waits until its local clock (synchronized to master) hits T to begin capturing data. Because all devices are sync'd to the master within a few milliseconds accuracy, this effectively aligns the start of recording across devices to a very tight margin (usually well below 50 ms difference, often within a few ms). The devices then proceed to timestamp their data relative to this common start. The PC also notes its own start time for any local recordings (like webcams) to align with the same T. During recording, the devices continue to exchange

sync information. Each status update from device to PC may include the device's current clock vs. the master clock (or implicitly, the PC knows when it sent a sync and what the device's last offset was). If any device's clock starts to drift beyond an acceptable tolerance (say more than a few milliseconds), the PC can issue a re-synchronization or simply record the drift for later correction. The synchronization engine might incorporate simple drift compensation — for instance, if one phone tends to run its clock slightly faster, the system can predict and adjust timing gradually (rather than waiting for a large error to accumulate). In this implementation, because the recording durations might be on the order of minutes to an hour, and modern devices have reasonably stable clocks, straightforward NTP-based periodic correction is sufficient to maintain sub-millisecond alignment.

Finally, the communication protocol assists synchronization by carrying timing info in every message. The JSON messages often include timestamps. For example, when an Android sends a preview frame to the PC, it tags it with the timestamp of frame capture; the PC can compare that with its own reception time and the known offset to estimate network delay and clock skew in real-time. Similarly, the PC's commands can carry the master timestamp. This pervasive inclusion of timestamps means that even if absolute clock sync had a small error, each piece of data can be re-aligned precisely in post-processing using interpolation or offset adjustment.

In summary, the **PC—Android communication** is realized via a reliable JSON/TCP socket protocol, enabling comprehensive remote control and live data streaming, while the **synchronization mechanism** ensures all devices operate on a unified timeline. Together, these allow the system to achieve a high degree of temporal precision: tests have shown the system tolerates network latency variations from $\tilde{1}$ ms up to hundreds of milliseconds without losing synchronization. This is accomplished by designing for asynchronous, non-blocking communication and by decoupling the *command* from the *execution* time (i.e., schedule actions in the future on a shared clock). The result is a robust coordination layer that underpins the multi-modal data collection with the necessary timing guarantees.

(Figure 4.6: Communication and synchronization sequence. This figure illustrates the sequence of interactions for device connection and a synchronized session start. Initially, each Android device connects to the desktop's socket server and sends a JSON handshake (including device ID and sensor capabilities). The desktop acknowledges and lists the device as ready. The figure then shows the synchronization phase: the desktop (master) sends a time sync request or NTP response to the phone, and the phone adjusts its clock offset. When the researcher clicks "Start" on the PC, the desktop broadcasts a StartRecording message with a specified start timestamp. All Android devices (and the PC's own data acquisition for webcams) wait until the shared clock reaches that timestamp, then begin recording simultaneously. During recording, devices send periodic Status messages (with current frame counts, battery, and time sync quality) and stream preview data (video frames, sensor samples) to the PC. The PC might send occasional Sync messages if needed to fine-tune clocks. Finally, on "Stop", the PC sends a stop command and each device halts recording and closes files, confirming back to the PC. This sequence diagram underscores how the protocol and sync mechanism work in tandem to coordinate distributed devices in time.)

1.5 4.5 Data Processing Pipeline

The **data processing pipeline** in the Multi-Sensor Recording System encompasses the steps from raw data capture to the production of analysis-ready outputs. It involves components on both the Android side (which perform on-the-fly processing and organization of data as it's collected) and the desktop side (which aggregates and post-processes data from all devices after or during a recording session). The design aim is to ensure that by the end of a session, all the heterogeneous data — video, thermal, GSR, etc. — are properly time-aligned, annotated, and stored in a structured format so that researchers can easily perform further analysis (such as feeding the data into machine learning models or statistical software).

On the **Android device**, the pipeline begins with the **sensor recorder components** described earlier. Each recorder not only captures data but may also perform lightweight processing. For instance, the Camera recorder uses the Camera2 API which can provide hardware-level video encoding; the app records video in MP4 files (with embedded timing metadata) and also extracts periodic still images or RAW images if needed for later calibration. The Thermal recorder processes each frame to compute temperature values (applying the camera's calibration and any emissivity settings) before writing the frame to disk; it also down-samples frames for preview streaming, which is a form of data reduction (only a subset of frames or a lower resolution image is sent live, while full data is stored). The Shimmer recorder, as discussed, formats the data into CSV lines and might compute basic derived metrics (for example, it could convert the raw GSR voltage to microsiemens using calibration constants, if not already done by the device driver). Additionally, the **HandSegmentation** module on Android runs a machine learning model on each video frame (or at a set frequency) to detect hand landmarks [**?**]. The output of that — e.g., coordinates of hand joints or a bounding box of the hand region — can be used to tag frames or even to save an additional data stream (like a timeline of "hand present or not" flags). By performing this on-device, the system reduces the amount of data that needs to be transmitted or stored (no need to save entire images for analysis of hand presence — just saving the coordinates or mask is enough, which is far smaller).

All data on the Android is saved in a **structured file system hierarchy** (often organized by session). For each recording session, the app creates a session folder containing the various files: e.g., `session_001_metadata.json` (with high-level info like session ID, start time, participant ID), `session_001_camera.mp4` (phone RGB video), `session_001_thermal.raw` (thermal binary file), `session_001_shimmer.csv` (physio data), etc. This local organization is part of the pipeline because it enforces consistent naming and indexing for later merging. The Persistence layer on Android ensures writes are flushed and files are closed safely at the end of sessions to avoid corruption.

Moving to the **desktop side**, during an active session, the desktop controller is receiving live previews and status, but it usually does not store all raw data coming over the network (since the high-quality data stays on devices until the session ends). However, the PC does record some information: it might save the low-resolution preview video or individual frames for a quick review (not research quality, but for reference), and it logs time-stamped events (e.g., "Recording started at T", "Device A battery low at T", etc.). Once the session is completed, the data from each device needs to be **consolidated**. In some configurations, the Android devices

might automatically upload their files to the PC (if a file transfer mechanism is enabled — this could be initiated by the PC requesting each device to send its files, possibly using the same socket or a separate channel). In other cases, a researcher may manually copy the files. Either way, the PC application provides tools to import the session data from all devices into a single location on the PC for analysis.

The desktop's **Data Processing components** then take over. A `DataProcessor` module on the PC can parse each data file (using knowledge of the format — for example, it knows how to read the thermal .raw file and extract frames and timestamps, or read the Shimmer CSV) and then perform multi-modal synchronization verification. Because all data streams were independently recorded, the system double-checks that the timelines align: it may, for instance, compare the timestamp of the first frame of the phone video with the master start time to compute an offset, and do the same for the first thermal frame and first GSR sample. Minor adjustments (order of tens of milliseconds) can be handled by shifting timestamps in software to perfect the alignment. This post-processing synchronization step is important if any device started a fraction of a second late or if there was clock drift — the Synchronization Engine on the PC assists by providing logs of the offset of each device over time, which the DataProcessor can use to correct timestamps. The result is that each piece of data can be assigned a **global timestamp** in a common reference (e.g., milliseconds since session start or an absolute UTC time).

Following synchronization, the pipeline can branch into different **data export and analysis preparation** tasks. A `DataExporter` component handles converting the data into formats needed for analysis. Commonly, researchers might want all sensor data in a single file or database, or in a form that can be loaded into Python or MATLAB for analysis. The system might generate a unified CSV or HDF5 file that contains synchronized timestamps and all sensor readings. For video data, it might extract per-frame timestamps and save them alongside the physiological signals. If needed, the video and thermal imagery can be merged — for example, some studies may overlay the thermal data on the RGB video frame or produce a side-by-side combined video; such an operation can be scripted using OpenCV with the calibration information (to map thermal pixels to the RGB frame if the cameras were calibrated). The Calibration Manager on the PC provides the calibration parameters (e.g., transformation matrices) to the DataProcessor for this purpose when needed.

Another aspect of the pipeline is **quality assurance (QA)**. The system includes a `DataSchemaValidator` and possibly a `QualityMonitor` that verifies the data collected meets certain integrity criteria. For instance, after recording, the software can check if the number of video frames roughly matches the expected frame rate and duration, or if there are gaps in the GSR timestamp sequence. It flags any anomalies (like dropped data or sensor errors) to the researcher. This is important for a thesis-grade project to demonstrate that the data is trustworthy. The QA results might be included in a session report automatically.

To facilitate iterative analysis, the desktop application also supports **post-session visualization**. The Files/Data tab can load a session's data and plot it (e.g., graph the GSR over time and allow overlaying markers where certain events happened, or scrub through the video with the corresponding thermal images). This isn't so much a part of the pipeline that creates

new data, but it helps in verifying and exploring the synchronized dataset.

In summary, the data processing pipeline ensures that raw streams from multiple sensors are first captured reliably (with minimal real-time processing except what's necessary for compression or region-of-interest extraction), then centrally synchronized and validated, and finally exported in a cohesive format. The pipeline leverages the structured approach of the system: each modality's data is handled by specialized code, but they converge in a common timeline. The design choice to timestamp everything and log rich metadata greatly simplifies the later stages of the pipeline, since the heavy lifting of alignment is mostly solved by design. This allows the **researchers to focus on analysis**, knowing that the incoming data has been properly collected and synchronized by the system. The pipeline thus transforms raw multi-modal data into an integrated dataset suitable for tasks like machine learning model training, statistical analysis of physiological responses, or visualization in publications.

(Figure 4.7: Data processing pipeline from data capture to analysis. The figure depicts the flow of data through various stages: at the left, raw data acquisition on each Android device (camera frames, thermal readings, GSR samples) along with initial processing (video encoding, thermal calibration, formatting of GSR values). These are saved as local files on the device. In the middle, the synchronization and aggregation step: the PC collects the metadata and possibly the data files from all devices, aligning them on a common timeline (using the master clock and timestamps). At the right, the output stage shows the generation of synchronized data outputs — for example, combined datasets, synchronized video playback with sensor overlays, and summary reports. Also indicated is a quality check loop, where the system validates data integrity (e.g., checking for missing frames or drift) and logs any issues. This pipeline ensures that by the end of this stage, all data is ready for the next chapter of evaluation or analysis.)

1.6 4.6 Implementation Challenges and Solutions

Building a complex multi-sensor system like this inevitably came with several implementation challenges. Throughout the development, we encountered issues related to synchronization, data volume, cross-platform integration, and sensor hardware quirks. This section outlines the key challenges and the solutions or design decisions we adopted to address them:

- **Precise Time Synchronization Across Platforms:** Ensuring that an Android phone and a PC (and possibly other devices) agree on time within a few milliseconds is non-trivial, given differences in operating system scheduling and clock stability. Our solution was to implement a **hybrid software NTP approach**. We ran a local NTP server on the PC and had the Android periodically sync to it, coupled with timestamped command protocols. By sending scheduled start times and using the PC as the single source of truth for time, we avoided the need for continuous tight coupling. We also added drift monitoring — if a device's clock started to stray, the system would either resync it or account for the offset in data post-processing. This approach yielded sub-millisecond synchronization accuracy in tests, meeting the project's requirements. An added benefit is that each device could operate independently if needed (in case of connection loss) and still later align via the timestamps, which gave us robustness against network issues [**?**].

- **Multi-Modal Data Integration and Volume:** Recording high-resolution video at 30 fps, thermal images at 25 fps, and GSR at 50 Hz simultaneously produces a vast amount of data per second. Handling this without data loss or overload was a challenge. We tackled it through **concurrency and efficient data handling**. On Android, each sensor recorder runs largely on its own thread or coroutine, writing to dedicated files or buffers so that no single thread becomes a bottleneck. We used optimized libraries (Camera2 with hardware codecs, buffered I/O streams for sensor data) to reduce CPU usage. Moreover, by performing some data reduction in real time (e.g., not every frame is forwarded as a preview, or sending compressed images), we kept the network throughput within reasonable limits. The system's modular architecture also helped: each data stream was independent, so if one modality temporarily lagged (say a burst of disk writes for video), it would not stall the others — they each had their own queues and threads. The outcome was a smooth integration where the data from different sources could be recorded in parallel reliably. Additionally, our **data schema** ensured integration after the fact: because every data point had a timestamp, we could merge streams offline without ambiguity. This design choice turned what could have been a complex synchronization problem into a straightforward data merge task using timestamps.

- **Bluetooth Reliability and Sensor Connectivity:** The wireless nature of the Shimmer GSR sensor introduced challenges like connection drops, signal interference, and the need to pair/manage Bluetooth in Android's constrained environment. The solution involved implementing robust **reconnection logic and buffering**. The ShimmerRecorder class was built to automatically detect a disconnect and retry connecting up to a few times before giving up [**?**]. We also made sure that short interruptions in connectivity did not result in data loss: if the Shimmer missed a few packets during reconnection, the system would fill the gap with a notation or interpolate later. We had to carefully manage Android's Bluetooth permissions and scanning (especially on newer OS versions that restrict background BT operations). This was solved by prompting the user upfront for the needed permissions and using the latest APIs for scanning/connecting that respect Android's power management. On the PC side, an alternative was ready: if a phone's Bluetooth failed, the researcher could connect the Shimmer directly to the PC's Shimmer Service, as a fail-safe. This dual-path approach improved reliability. Finally, to mitigate interference and ensure a strong signal, we advised that the phone be kept near the Shimmer sensor during recordings (something noted in the user guide).

- **Cross-Platform Integration and Compatibility:** Developing and debugging two separate applications (Android and Python) that must work in concert posed compatibility issues — differences in programming languages, data serialization, and even how each handles threading. A specific example was ensuring that the JSON protocol was interpreted exactly the same on both ends, and that special data types (like binary image frames or high-precision timestamps) survived the journey. We addressed this by **standardizing the communication and using well-tested libraries**. Python's use of the `json` library and Android's use of Kotlin's JSON handling (or manual parsing) were aligned by

a strict schema: we defined, in documentation, every message's format and wrote unit tests for encoding/decoding on both sides. We also decided on using UTC timestamps in milliseconds everywhere to avoid any time zone or locale issues. Another aspect was thread coordination: the Android app uses Handler threads and coroutines, while the PC uses Qt's QThreads and async I/O. If the PC sent multiple commands quickly, we had to ensure the phone could queue and process them properly without race conditions. The fix was to implement a simple **acknowledgment system** — the PC would wait for an ACK of a command before sending the next (for critical commands), or tag messages with sequence numbers so out-of-order processing could be detected. Through such measures, we achieved a reliable cross-platform partnership between the apps. Continuous integration testing, where we ran both the Android and Python components in test scenarios, helped catch incompatibilities early.

- **Resource Constraints and Performance Optimization:** Running intensive tasks (recording video, processing images, streaming data) on a mobile device for extended periods can lead to performance degradation or even crashes due to memory, CPU, or thermal constraints. We encountered issues like the phone's CPU heating up and throttling during long sessions, or the garbage collector pausing the app if too much memory was used improperly. Our solution was two-fold: **optimize and monitor**. We optimized by using efficient data structures and avoiding unnecessary copies of data (for instance, reusing byte buffers for thermal frames rather than allocating new ones each time). We also leveraged lower-level APIs when possible (e.g., using Android's native media codec for video instead of a heavy software encoding). On the monitoring side, we built a **Performance Monitoring layer** in the app that tracks memory usage, CPU load, and frame processing time. If any metric exceeded a threshold (for example, if frame processing was taking too long and queue lengths were growing), the system would log it and could adjust behavior (like dropping preview frames to catch up). Additionally, we exposed some of these stats on the PC UI so the user could see if a device was struggling. With these strategies, we managed to keep the system running within the devices' capabilities — e.g., an Android phone could run a 20-minute session with thermal and video without overheating by dynamically lowering preview frame rate if temperature rose, which we implemented as a simple adaptive measure.

In conclusion, each major challenge was met with a targeted solution that was integrated into the system's design. The emphasis on modular architecture greatly facilitated this: we could improve or fix one part of the system (say, the Bluetooth reconnection logic) without needing to overhaul unrelated parts (like the video recorder). This flexibility allowed iterative refinement. Many of these challenges, especially synchronization and multi-threaded performance, are common in distributed sensing systems; our implementation demonstrated effective strategies by combining well-known techniques (like NTP time sync, buffering, multithreading) with custom engineering (like our JSON command protocol and cross-checking of timestamps). The result is a robust system where all components work together smoothly despite the complexities involved, providing high-quality, synchronized data for the research objectives. Each solution reinforced the system's reliability and validated the chosen design principles in a real-world setting.