ChatGPT

Monorepo Integration Blueprint: fyp-gsr-windows & fyp-gsr-android

Phase 1: Monorepo Setup & Shared Infrastructure

High-Level Goals: Establish a unified project structure in a single repository using IntelliJ IDEA, integrating both the Python/PySide6 desktop app and the Kotlin/Android mobile app. Set up all foundational tools and protocols (Gradle for Android, Poetry for Python, Protocol Buffers, Lab Streaming Layer) and ensure both apps build and run in the new environment.

Development Subtasks (Step-by-Step):

1. **Repository Structure & Version Control [Core]:** Create a monorepo directory with clear subfolders for each component. For example:
2. `android-app/` (Android Studio project for the Kotlin app)
3. `pc-app/` (Python PySide6 application code)
4. `proto/` (common Protocol Buffer definitions)
5. any shared docs or config at the root. Initialize Git (if not already) and add appropriate `.gitignore` (e.g., ignore `*.idea/`, build artifacts, virtualenvs, etc).

```
fyp-gsr-monorepo/
├── android-app/
│   ├── app/                    # Android app module (Kotlin code, resources)
│   ├── build.gradle            # Gradle build for app
│   └── ... (Gradle wrapper, settings.gradle, etc.)
├── pc-app/
│   ├── fypgsr/                 # Python package for the desktop app
│   │   ├── __init__.py
│   │   ├── main.py             # Entry point for PySide6 GUI
│   │   └── ... (other modules)
│   ├── pyproject.toml          # Poetry configuration for Python deps
│   ├── poetry.lock
│   └── venv/                   # (or manage venv externally; not in VCS)
└── proto/
    └── messages.proto          # Protocol Buffer definitions for commands &
data
```

1. **IntelliJ Multi-Module Project Configuration [Core]:** Open IntelliJ IDEA (preferably Ultimate for combined Java/Python support) and set up a single project containing both modules:
2. Import the `android-app` as a Gradle project (IntelliJ will detect the Android project structure).
3. Add a Python module/facet for the `pc-app` directory. In IntelliJ, right-click the project or `pc-app` folder and choose “Add Framework Support... -> Python” [1] to enable Python support with the Python plugin. Ensure the Python SDK (e.g., a venv or system interpreter) is configured for this module.

4. Verify the project structure in **File > Project Structure**: the Android module should use an Android SDK, and the Python module should have a Python SDK/Interpreter set. The modules remain separate in build logic but share the same repo and can be edited in one IDE window.
5. Set up run configurations:
    - **Android App Run Config:** created automatically when you import the Android module. Configure a device/emulator deployment if needed.
    - **Python App Run Config:** create a new run configuration for Python, pointing to `pc-app/fypgsr/main.py` (or whatever the GUI launcher script is), using the Python interpreter (venv) you set. This allows launching the PySide6 app from IntelliJ.

6. (Optional) Define a **Compound** run configuration if you want to launch both apps together for testing (one config can deploy Android app and another starts Python app).

7. **Tooling Setup – Android Gradle [Core]:** In `android-app/build.gradle` (and/or module's `build.gradle`), configure necessary plugins and dependencies:

8. Apply the `com.android.application` plugin and Kotlin Android plugin as usual.

9. Add the **Protocol Buffers Gradle plugin** (`id "com.google.protobuf"`). In the Gradle `dependencies` block, include the ProtoBuf compiler and runtime:

```
buildscript {
    dependencies {
        classpath "com.google.protobuf:protobuf-gradle-plugin:0.8.19"
    }
}
plugins {
    id "com.google.protobuf" version "0.8.19"
    ...
}
dependencies {
    implementation 'com.google.protobuf:protobuf-javalite:3.21.12'  //
use lite runtime for Android [2] [3]
    implementation 'edu.ucsd.sccn:liblslAndroid:
1.16' // (if an official Maven artifact exists; if not, will use .aar in
libs)
    // ... other deps (e.g. AndroidX, etc.)
}
protobuf {
    protoc { artifact = "com.google.protobuf:protoc:3.21.12" }
    generateProtoTasks { all().forEach { task ->
        task.builtins { java { option "lite" } }  // use lite proto
classes
    } }
}
```

This instructs Gradle to generate Java (lite runtime) classes from `.proto` definitions during build. Ensure **IntelliJ is set to delegate builds to Gradle** so that codegen runs automatically [4]. In IntelliJ settings: *Build, Execution, Deployment* > *Build Tools* > *Gradle* > *Runner* > *check "Delegate IDE build/run actions to Gradle"*.

10. **Lab Streaming Layer (LSL) Integration (Android):** Obtain the LSL library for Android:
    - Easiest method: build or download the `liblsl-Java` package. The LSL project can produce an AAR containing the native libraries and Java wrapper [5]. Follow LSL's official guide: clone the LSL repo and run the gradle build to generate `liblsl-Java-Release.aar` [6].
    - Add the AAR to your project: copy it into `android-app/app/libs/` and add `implementation files('libs/liblsl-Java-Release.aar')` in Gradle. This avoids having to manually handle `.so` files, as the AAR includes them for various architectures [7].
    - Alternatively, if a precompiled AAR or Maven dependency is available (e.g., `"edu.ucsd.sccn:liblsl:1.16"`), use that.
    - After adding LSL, sync Gradle and ensure the project builds. You will use the LSL Java API (accessible from Kotlin) to create outlets/inlets on Android.
11. Other Android tooling: Ensure you have the **NDK installed** if needed for LSL (the AAR likely contains prebuilt binaries, but if you build from source, NDK setup is required as per LSL docs). Also add any **Shimmer SDK** or Bluetooth libraries if the app interfaces with Shimmer sensors (e.g., include Shimmer's BLE API .aar or .jar if provided).

12. **Tooling Setup – Python/Poetry [Core]:** In `pc-app/pyproject.toml`, configure the Python project:

13. Define the package (e.g., name `fypgsr`) and dependencies:

```
[tool.poetry]
name = "fyp-gsr-windows"
version = "0.1.0"
packages = [{ include = "fypgsr" }]
[[tool.poetry.dependencies]]
Python = ">=3.9"
PySide6 = "^6.5"
pylsl = "^1.15"      # Python LSL library for inlet/outlet
protobuf = "^4.23"   # Google protobuf for Python (for parsing command
messages)
```

14. Install dependencies by running `poetry install` (or use a venv + pip if not using Poetry). This will set up PySide6 (for GUI), pylsl, etc.
15. Create an entry-point script (if not already) in `fypgsr/main.py` that launches the PySide6 GUI. In Poetry, you can also define a console script entry point for `fypgsr` if desired, but since we'll likely freeze the app with PyInstaller, that's optional.

16. Verify you can run the Python app standalone (e.g., `poetry run python -m fypgsr.main`). Initially it might just open a window or log something.

17. **Define Protocol Buffer Schemas [Core]:** Design a `.proto` file (e.g., `proto/messages.proto`) to define unified message formats for sensor data and commands. This ensures type-safe communication between Android (Kotlin/Java) and Python:

18. Example schema (proto3):

```
syntax = "proto3";
package fypgsr;
message GsrSample {
  string device_id = 1;
  float value = 2;
  double timestamp = 3;
}
message Command {
  enum Type {
    START_STREAM = 0;
    STOP_STREAM = 1;
    START_CAMERA = 2;
    STOP_CAMERA = 3;
    MARK_EVENT = 4;
  }
  Type type = 1;
  string target_device = 2;  // which phone to apply to, or "*" for all
  string payload = 3;        // optional parameters (could be JSON or
simple values)
  double timestamp = 4;      // command issue time (if needed)
}
```

19. Add any other messages as needed (e.g., a `CameraFrame` message if sending images, though large binary data might be handled outside of proto).
20. **Code generation:** Use Gradle to generate the Kotlin/Java classes for these messages (the plugin will create Java classes in `build/generated/` which Kotlin can use). For Python, generate code by running `protoc` manually or via a script:
    - Install protoc compiler (or use the one in Gradle if accessible). Then from repo root, run a command like:

      ```
      protoc --proto_path=proto --python_out=pc-app/fypgsr proto/
      messages.proto
      ```

      This will produce `messages_pb2.py` in the `fypgsr` package. (Alternatively, use `poetry run python -m grpc_tools.protoc ...` if `grpc_tools` installed.)
    - Include the generated `messages_pb2.py` in the repo so that the Python app can import it. (Every time you update the `.proto`, regenerate both Java and Python code.)
21. Verify the generated code on both sides. In Python, you should be able to `import fypgsr.messages_pb2 as proto` and construct, e.g., `proto.Command()` objects. In Kotlin, classes like `GsrSample` and `CommandOuterClass.Command` will be available (depending on package structure).

22. **Establish Base Communication Interfaces [Core]:** Even if we haven't implemented full functionality, set up skeleton classes and functions in both apps that will later handle data streaming and commands:

23. **Android side:** Create (or identify) a component (e.g., a singleton object or Android Service) to manage sensor data collection and LSL streaming. For example:

```
object LslStreamManager {
    private lateinit var gsrOutlet: LSL.StreamOutlet
    fun initStreams(deviceId: String) {
        // Initialize LSL outlet for GSR data
        val info = LSL.StreamInfo("GSR_$deviceId", "GSR", 1, 10.0,
LSL.ChannelFormat.float32, "gsr-$deviceId")
        gsrOutlet = LSL.StreamOutlet(info)
        // (In the future, also init command inlet, etc.)
    }
    fun pushGsrSample(value: Float, timestampSeconds: Double? = null) {
        if (::gsrOutlet.isInitialized) {
            val sample = floatArrayOf(value)
            gsrOutlet.pushSample(sample, timestampSeconds ?:
LSL.local_clock())  // use LSL clock timestamp
        }
    }
    // Placeholder for receiving commands (to be implemented in Phase 3)
    fun onCommandReceived(cmd: fypgsr.Command) { /* stub */ }
}
```

Here `LSL` is the Java wrapper package from liblsl. We use a monotonic clock timestamp from `LSL.local_clock()` when pushing samples to ensure synchronization [8]. The stream info includes a unique source ID (`"gsr-$deviceId"`) so multiple devices are distinguishable.

24. **Python side:** Create a module or class to handle incoming LSL streams. For example, a `DataReceiver` class:

```
import pylsl
from fypgsr import messages_pb2
class DataReceiver:
    def __init__(self):
        self.inlet = None
    def connect_to_stream(self, name: str = "GSR_*"):
        streams = pylsl.resolve_byprop("name", name, timeout=5)
        if streams:
            self.inlet = pylsl.StreamInlet(streams[0])
    def read_sample(self):
        if self.inlet:
            sample, timestamp = self.inlet.pull_sample(timeout=0.0)
            return sample, timestamp
```

This simplistic example resolves a stream (by name or other criteria) and can pull samples. In practice, you might run a loop in a background thread or use callbacks to continuously read incoming data.

25. At this phase, these classes may be mostly placeholders or basic implementations. The goal is to have the structure ready for Phase 2 when we actually stream real data.

26. **IntelliJ Run/Test Baseline [Core]:** At the end of Phase 1, verify that:

27. The Android app builds (via Gradle) and can run on a device (it might just show a basic UI or even a blank activity for now). Ensure no runtime errors from missing libs (the LSL lib may need testing by calling a simple `LSL.local_clock()` in code to confirm it doesn't crash).
28. The Python app launches (perhaps a simple PySide6 window with a label like "Hello, FYP GSR"). In IntelliJ, you should be able to start the Python run config and see the GUI.
29. The proto integration doesn't break anything: e.g., importing `messages_pb2` in Python works, and using `Command.newBuilder()` in Kotlin works (you might write a quick unit test or a log to ensure proto objects can be created on both sides).

File/Module Structure & Naming Conventions: Use clear, consistent naming: - For Python, use a package name (e.g., `fypgsr`) matching the project. Organize modules by feature (e.g., `lsl_comm.py` for LSL communications, `ui/` for UI code, etc.). Follow PEP8 for naming (modules_lowercase, ClassNames in CamelCase, etc.). - For Android, follow typical Kotlin conventions: package name (e.g., `com.yourname.fypgsr`), use CamelCase for classes (`MainActivity`, `GsrService`, etc.), and lowercase for resource files. Group code by functionality (maybe a `sensors` package for sensor logic, `network` for LSL/commands logic, etc.). - Keep the proto definitions in a neutral location (`proto/messages.proto`). The generated classes will reside under `build/generated` in Android and under `fypgsr/messages_pb2.py` in Python. Ensure these are not modified manually – treat the `.proto` as the source of truth.

IntelliJ IDEA Configuration Notes: - After adding the Python facet, if IntelliJ highlights imports (like PySide6 or pylsl) as unresolved, configure the Python SDK and mark `pc-app` as source root. Also, verify that `pc-app/fypgsr` folder is marked as a "Sources" folder (right-click > Mark Directory As > Sources Root). - Use IntelliJ's **Services** or **Run Dashboard** to manage running both apps if needed. This can help monitor logs from both concurrently. - Set the **Project SDK** to Java (for Android) and **Module SDK** for Python separately. IntelliJ will allow both to coexist. If you encounter issues like one language highlighting breaking when switching SDK, ensure the Python facet is attached to the Python module and not affecting the Android module. - Consider using **virtual environments** for Python (Poetry will typically create one). You can point IntelliJ's Python SDK to the Poetry-created venv (find it via `poetry env info`). This keeps dependencies isolated.

Estimated Effort/Priority: Phase 1 is **core** and foundational. It might take ~1 week to fully set up given tool configuration and debugging, especially the LSL Android integration (which can be tricky). Nearly all tasks here are high-priority, as they unblock later phases. The only optional tasks might be setting up nice-to-have IntelliJ features (like compound run config), which can be deferred.

Testing & Acceptance Criteria: - **Configuration Verification Test:** Ensure that a fresh clone of the repository can be opened in IntelliJ and both modules build successfully. (e.g., a colleague should be able to follow README instructions to set up the environment, run `gradle build` for Android, and `poetry install` + `poetry run` for Python without issues.) - **Hello World Run Test (Manual):** Run the Android app on a device – it should install and launch (even if just a basic UI). Run the Python app – it should open a window. This confirms the monorepo and IDE setup is correct. - **Proto Codegen Test (Unit):** Write a quick unit test on each side to instantiate and serialize/deserialize the protobuf messages: - In Python, create a `Command` object, serialize to bytes and parse back, assert the fields remain correct. - In Kotlin, do similar using the generated Java classes (or simply log the object to ensure it can be created). - This ensures the proto definitions are accessible and working in both environments. - **LSL Library Test (Unit/Integration):** On Android, call `LSL.local_clock()` and log the result, or create a dummy `StreamOutlet` and push a sample; on Python, use `pylsl.local_clock()` similarly. No exceptions or crashes should occur. (This might be done in Phase 2 as part of deeper testing, but even at setup, a quick check is useful.) - **IntelliJ Config Check:** Verify that pressing "Run" on both configurations in IntelliJ actually launches the apps with the correct

interpreter/SDK. If any misconfiguration (e.g., wrong Python path) exists, fix it. This is more of a manual sanity check.

By the end of Phase 1, the development environment is ready with a maintainable structure, and we have a clear runway for implementing functionality in subsequent phases.

Phase 2: Sensor Data Acquisition & Streaming via LSL

High-Level Goals: Implement end-to-end sensor data streaming from the Android device to the PC application using the Lab Streaming Layer. This involves reading the GSR sensor on Android (via Shimmer API or device sensors), pushing those samples out over LSL, and receiving them in the Python app for display or recording. The focus is on a single device->PC pipeline for now. Additionally, ensure timestamps are handled properly at the source.

Development Subtasks:

1. **Android Sensor Integration [Core]:** Implement the code to acquire GSR measurements on the Android app:
2. If using a **Shimmer GSR device**: integrate the Shimmer SDK or Bluetooth communications. Likely, the Shimmer streams raw packets to the phone via Bluetooth. Write a `ShimmerService` or similar class to connect to the sensor, subscribe to GSR data, and decode it. For example, if Shimmer provides a callback with raw byte packets, parse those into meaningful GSR values (in µS or appropriate units). *Unit test idea:* Use recorded raw data bytes to verify your parsing function outputs expected values (see Testing below).
3. If the GSR is read from an Android sensor or a simulated source, implement accordingly (for Shimmer, this is probably via their API).
4. Ensure this data acquisition runs on a background thread or service, not on the main UI thread. You might use a Foreground Service if the app needs to collect data in background or when screen is off.

5. Example (pseudo-code):

```
class ShimmerGsrManager {
    fun connect(deviceAddress: String)
{ /* connect via BT, subscribe to data */ }
    fun onDataReceived(packet: ByteArray) {
        val gsrValue = decodeGsr(packet)
        LslStreamManager.pushGsrSample(gsrValue)
    }
    private fun decodeGsr(packet: ByteArray): Float {
        // implement conversion using Shimmer spec (e.g., parse bytes to
int, apply scale)
    }
}
```

   It's useful to keep the decoding logic separate and unit-testable.

6. **Android LSL Outlet for GSR [Core]:** Use the `LslStreamManager` (from Phase 1) to push the sensor readings to LSL:

7. Finish implementing `LslStreamManager.initStreams()`: set stream name and metadata. For example, `StreamInfo(name="GSR", type="GSR", channel_count=1, nominal_srate=ShimmerGsrManager.SAMPLE_RATE, channel_format=ChannelFormat.float32, source_id=deviceId)`. You can include additional meta-data (LSL allows embedding XML) such as device type, unit, etc., but that's optional.
8. Ensure `pushGsrSample(value)` is called whenever a new sensor sample is ready. Pass a timestamp:
    - Ideally use `LSL.local_clock()` at the moment of data arrival from sensor (this is a monotonic clock synced across devices by LSL [8]).
    - Alternatively, if the sensor data comes with its own timestamp, you could use that if it's in system time; but likely easier to use LSL clock for consistency.
9. Test on the device: run the app connected to a Shimmer sensor (or simulate data if you can). Monitor logcat to ensure samples are being pushed (you can log each push or count of pushes).
10. **Performance note:** If the sensor produces data at a high rate, consider pushing samples in chunks (LSL allows `push_chunk`) to reduce overhead. But GSR is typically low-frequency (e.g. 10-50 Hz), so pushing one by one is fine.

11. **Python LSL Inlet for GSR [Core]:** In the desktop app, implement continuous data reception:

12. Expand the `DataReceiver` (or create a dedicated `GsrInlet` class) to find and subscribe to the GSR stream:

```
class GsrInlet:
    def __init__(self, device_id: str = None):
        # Resolve by device if given, else get any GSR stream
        query = f"name='GSR'" + (f" and source_id='{device_id}'" if
device_id else "")
        streams = pylsl.resolve_streams(wait_time=2.0)

# filter streams manually since pylsl doesn't support full query strings
        streams = [s for s in streams if s.name() == "GSR" and
(device_id is None or s.source_id() == device_id)]
        if not streams:
            raise RuntimeError("No GSR LSL stream found")
        self.inlet = pylsl.StreamInlet(streams[0])
        self.inlet.set_postprocessing(0)  # no post-processing (or
LSL.POST_NONE)
    def pull_sample(self):
        sample, ts = self.inlet.pull_sample(timeout=1.0)
        return sample, ts
```

    This will grab one sample at a time. In practice, you will run a loop or a QTimer (since PySide6 GUI) to periodically call `pull_sample` and update the UI.
13. **UI Update:** Design how the GSR data will be shown. For instance, you might display the numeric value or plot a real-time graph. Using PySide6, you could integrate a graphing library (or simple QPainter on a QWidget) for plotting. Initially, to keep it simple, show the latest GSR value and maybe the stream rate.

14. It's critical to not block the GUI thread while reading data. Solutions:
    - Use a separate thread (Python `threading.Thread`) that continuously reads from `GsrInlet` and emits signals to the Qt main thread to update UI.
    - Or use QTimer with a small interval (e.g. 100 ms) to poll `pull_sample()`. Pylsl's pull is thread-safe, but be mindful of Python GIL for performance if rates are high.

15. **Timestamp handling:** Each pulled sample comes with a timestamp (likely in *local time of the PC** after offset correction by LSL). Confirm that timestamp is reasonable (e.g., use `pylsl.local_clock()` to compare). You might not display the timestamp to user, but you will use it for syncing if needed. For now, just ensure you capture it (e.g., store it along with the data if plotting, to get correct X axis).

16. **Verification of Data Flow [Core]:** Once the above is implemented, perform an integration test on the system:

17. Launch the Android app (with sensor connected) and the Python app (ensure both on same network, LSL uses multicast to discover).
18. The Python app should discover the GSR stream automatically (within a second or two). Implement some visual cue: e.g., a status label "Connected to GSR stream [device_id]" when found. If not found after a timeout, inform the user ("No GSR stream found").
19. When data is flowing, confirm that the values on the PC match expected patterns. If you have a way to cross-check (e.g., Shimmer might also display on its own app or a known range), verify the magnitude. If the decoding is wrong (e.g., off by a factor or offset), adjust decode logic.
20. Check the **latency**: LSL on a local network should have very low latency (tens of milliseconds). You can test this by perhaps abruptly changing the sensor input (e.g., touch the GSR electrodes to see a spike) and observing the PC reaction time. It should be nearly instantaneous to a human observer.

21. In code, you can measure latency by comparing `pylsl.local_clock()` at receipt vs the sample's timestamp (which LSL aligns to sender clock). The difference minus network offset should be small (LSL's internal sync aims for sub-millisecond accuracy on LAN [9]).

22. **Data Logging (Optional):** As an extension, implement a simple logging of the incoming data on the PC:

23. For example, allow the user to start/stop a recording that writes timestamps and GSR values to a CSV file or, ideally, to an XDF file (the LSL recording format) if you integrate with LabRecorder or an XDF library. This can be useful for offline analysis and verifying synchronization later.
24. This might be optional at this stage, but keep in mind for the final deployment (so experimenters have raw data saved).

Testing & Acceptance Criteria:

- **Unit Test – GSR Decoding:** If using Shimmer, create unit tests for the byte decoding function. For example, use known raw byte sequences from Shimmer documentation and assert that the output value in Siemens is as expected. (This might require known calibration constants; use Shimmer's reference or a snippet from their SDK docs for expected values.)
- **Integration Test – LSL Stream Discovery:** Write a test (could be an instrumentation or manual test) to ensure the PC finds the Android's LSL outlet:
- On Android, start streaming (perhaps have a UI button "Start Stream" that triggers `LslStreamManager.initStreams` and sensor connection).

- On PC, attempt to resolve the stream. You can automate this by writing a small script that uses `pylsl.resolve_byprop("name","GSR", timeout=5)` and ensure it returns a stream. This can be part of an integration test suite (run both apps in a test environment if possible, or use a fake outlet).
- Acceptance: The stream is resolvable within a few seconds at most.
- **Integration Test – Data Accuracy:** Compare data from the phone vs PC:
- If possible, log a few samples on Android side (e.g., print the values) and simultaneously log what PC receives. They should match (within any expected noise/quantization).
- Ensure timestamps are consistent: for a given sample, if you compute the difference between PC receive time and the embedded timestamp, the network transit delay should be small (couple of ms). LSL internally measures and compensates clock offsets similar to NTP [10], so the timestamp should be the phone's clock, corrected for offset.
- **Manual Test – Real-time Display:** Have a user operate the system:
- Start the PC app, ensure it's ready to receive (maybe show "waiting for stream").
- Start the Android app, connect sensor, and start streaming.
- Observe on the PC a live readout or graph of GSR. Manipulate the sensor (e.g., touch to increase conductance or remove to drop) and see the PC reflect these changes promptly.
- Stop the sensor stream (perhaps by a button or just closing the app) – PC should detect loss of stream (you can catch the exception or `None` returns from `pull_sample` after a timeout) and update UI ("Stream disconnected").
- **Robustness Test:** Let the system run for an extended period (say 10 minutes) to ensure no memory leaks or crashes. The PC graph should continue updating and not drift noticeably (baseline should remain aligned since timestamps are continuously synced by LSL's protocol).

Acceptance Criteria: By the end of Phase 2, the system should reliably stream GSR data from one Android device to the PC in real-time. The PC application displays or logs the data in a meaningful way. The data is timestamped at the source (phone) and maintains time synchronization within a few milliseconds between devices (leveraging LSL's built-in clock sync similar to NTP [10]). Any decoding of sensor data is verified for correctness. This phase delivers the core functionality of synchronized biosignal acquisition.

Phase 3: Remote Control Commands & Camera Integration

High-Level Goals: Enable the PC application to remotely control aspects of the Android app, such as starting/stopping data streams and triggering the phone's camera. Introduce a bidirectional communication channel using a common protocol (likely via LSL or an alternative) for command messages defined in the proto schema. Implement the phone's response to commands (especially camera capture functionality). Ensure all actions are synchronized and logged with timestamps.

Development Subtasks:

1. **Command Communication Channel [Core]:** Establish a mechanism for the PC to send commands and the phone to receive them (and possibly acknowledgments back):
2. **Using LSL:** Implement a **Command Outlet** on the PC and a **Command Inlet** on the Android:
    - On the PC side (Python), create a `pylsl.StreamInfo` for commands, e.g. `info = pylsl.StreamInfo("GSRCommands", "Command", 1, 0, pylsl.cf_string, "cmd-stream")`. Here `channel_count=1` and we use `cf_string` (string format) for simplicity to send command data (could also use cf_int8 and pack bytes). Open a `StreamOutlet` with this info. This outlet will be used to push command messages.

- On the Android side, resolve this stream and open a `StreamInlet`. Because now the PC is the producer and phone the consumer, you might need to start the PC's outlet first or design a handshake.
- Alternative: The phone could announce it's ready and PC then opens outlet, or PC opens on startup and phone keeps trying to resolve. LSL discovery is cheap, so polling is fine.
- **Protocol usage:** When sending a command, the PC should create a `Command` protobuf message (from `messages_pb2.Command`), fill the fields (type, target, etc.), then serialize to a bytes string. If using `cf_string`, encode the bytes to e.g. base64 or hex, or simply send as UTF-8 if it's safe (raw bytes may not pass as string). Another approach is to use `cf_int8` with channel_count equal to byte length and send the byte array – but LSL requires a fixed channel count. Simpler: use string and base64.
- For example:

```
import base64, pylsl
from fypgsr import messages_pb2
class CommandSender:
    def __init__(self):
        info = pylsl.StreamInfo("GSRCommands", "Command", 1, 0,
pylsl.cf_string, "cmd-stream")
        self.outlet = pylsl.StreamOutlet(info)
    def send_command(self, cmd: messages_pb2.Command):
        # Serialize and encode to base64 for safe string
transmission
        data_bytes = cmd.SerializeToString()
        data_b64 = base64.b64encode(data_bytes).decode('ascii')
        self.outlet.push_sample([data_b64])
```

On the Android side, use `StreamInlet` to receive, then decode:

```
val results = commandInlet.pullSample()  // returns array of
strings
if (results != null) {
    val b64 = results.first() as String
    val cmdBytes = Base64.decode(b64, Base64.DEFAULT)
    val cmd = Command.parseFrom(cmdBytes)  // using protobuf-
generated class
    handleCommand(cmd)
}
```

This approach ensures we use the proto for structured commands while leveraging LSL for transport.

3. **Alternate Approach (Optional):** Use direct network sockets for commands. For instance, open a TCP or UDP socket between PC and phone for control messages. This could allow binary proto messages directly. However, given LSL is already in use and ensures sync, it's simplest to use LSL for commands as well to avoid maintaining two different networking methods. We stick with LSL unless unforeseen issues arise.

4. **Defining Command Types & Actions [Core]:** Decide on the set of remote commands needed (already outlined in proto `Command.Type`). Implement the PC UI to send these commands and the Android logic to handle them:

5. Common commands might include:
    - **START_STREAM / STOP_STREAM:** Start or stop the sensor data stream on the phone. (E.g., PC user presses “Begin GSR”, PC sends START_STREAM, phone begins collecting and pushing data. In Phase 2 we might have started streaming immediately; now we can gate it until commanded.)
    - **START_CAMERA / STOP_CAMERA:** Start or stop camera recording or streaming on the phone.
    - **MARK_EVENT:** Place a timestamped marker (could be used to mark an experimental event – the phone might not need to do much except possibly log the marker).
6. On the **PC side (UI)**: Add controls, e.g., buttons or menu options for these actions. For instance, a "Start Recording" button triggers both START_STREAM and START_CAMERA commands (if we want to begin both sensor and video together), or separate controls if needed. A "Mark Event" button could send a marker at important moments.
7. When a control is used, the PC creates the Command message: set the type (e.g., `cmd.type = Command.START_CAMERA`), set `target_device` (for now, maybe the single device ID or just a wildcard if one phone), set any payload (maybe not needed for basic commands), and set `timestamp` (optional, could use PC’s time or leave for phone to timestamp when received).

8. Then `CommandSender.send_command(cmd)` is called.

9. On the **Android side**: Implement a handler function `handleCommand(cmd: Command)` (as hinted in code above) to execute the appropriate action:

```
fun handleCommand(cmd: Command) {
    when (cmd.type) {
        Command.Type.START_STREAM -> {
            ShimmerGsrManager.connectIfNeeded();
            LslStreamManager.initStreams(deviceId); // start streaming
sensor
        }
        Command.Type.STOP_STREAM -> {
            ShimmerGsrManager.disconnect();
            LslStreamManager.closeStreams();
        }
        Command.Type.START_CAMERA -> startCameraRecording()
        Command.Type.STOP_CAMERA -> stopCameraRecording()
        Command.Type.MARK_EVENT -> markEvent(cmd.timestamp)
    }
}
```

    - The above assumes `Command` is the generated proto class available in Kotlin (you’d access it via something like `CommandOuterClass.Command` depending on how protos are generated; consider writing wrapper functions for convenience).
    - `markEvent(timestamp)` could simply log the event or use LSL’s event markers. An approach: one can create a separate LSL stream for “Markers” and push an event with

timestamp (some LSL setups do that for experiment events). But to simplify, maybe this command's presence itself is enough (or PC will mark events on its side when sending).
    - The camera functions we detail next.

10. **Camera Control and Data Handling [Core]:** Implement using the Android camera to capture data in response to PC commands:

11. Decide between **recording video vs streaming frames**:
    - For research, recording high-quality video on the phone (and timestamping it) is often preferable (network streaming of video can be bandwidth-heavy and may not be needed if the goal is synchronization – you can sync via timestamps later). So, **enable raw video recording on the phone** when commanded [11]. The PC might just get a notification or low-res preview.
    - Optionally, provide a preview stream (like a low FPS, low resolution image sequence) to the PC for monitoring.
12. **Android Implementation:**
    - Use CameraX or Camera2 API for ease. For example, with CameraX, set up a VideoCapture use case.
    - When `startCameraRecording()` is called (from a START_CAMERA command):
    - Prepare a file path (e.g., in app storage or SD card) with a timestamped filename (to later correlate with data, include deviceId and start time in the name).
    - Start recording via CameraX's `videoCapture.startRecording(outputOptions, ...)` or MediaRecorder API. Alternatively, if high-resolution and stability are needed, use the native camera app via Intent (less control though).
    - If providing preview to PC: you could also set up an ImageCapture or Preview use case and obtain frames (ImageAnalysis use case) periodically. For example, using ImageAnalysis, get a frame every X seconds, compress to JPEG, then send via an LSL outlet (similar to how we send commands, but probably as a separate stream like "CameraPreview" with type "video" or "image" and content as a JPEG byte array).
        - This is optional/advanced; if doing it, throttle to maybe 1 fps to avoid flooding.
    - When `stopCameraRecording()` is commanded:
    - Stop the recording (end the video file and close it properly).
    - Possibly notify the PC that recording stopped (could send a confirmation command back or at least update some status in UI on next data refresh).
    - Ensure proper permission handling for camera and storage on Android (request camera permission at runtime if not already).
    - **Timestamping:** Mark the start time of recording (perhaps embed in the video metadata by starting exactly at an LSL timestamp or record system time). If the phone and PC clocks are synced via NTP or LSL, you can later align video and sensor data. For example, log an event like "CameraStart at [LSL time]" in the data stream as well (maybe push a marker sample when camera starts).
13. **PC Handling:**
    - If not streaming video frames, the PC can simply display a status like "Camera recording started on Device X" when it sends START_CAMERA. It could enable a timer or indicator during recording.
    - If streaming preview frames via LSL:
    - Similar to GSR, set up an LSL inlet for "CameraPreview" stream. When frames (likely as byte array or base64 string) arrive, decode (to image) and display in the UI (perhaps in a QLabel as pixmap).

    - This can be heavy; ensure decoding is done off the main thread if frequent. However, at 1 fps it should be fine.
    - *Testing note:* Network bandwidth for images: e.g., a 320x240 JPEG could be ~30KB, at 1 fps that's fine. Avoid full HD frames live over LSL.
14. **Synchronization considerations:** If multiple data streams (sensor vs video) are recorded, rely on timestamps to sync. The GSR data is timestamped via LSL; the video file has its own internal timestamps (frame times). You might later sync by matching the "CameraStart" event timestamp with the video's first frame time (if clocks were NTP-synced or using LSL time for event marker).

15. Provide user feedback on PC: e.g., if user clicks "Start Camera", maybe disable that button and show "Recording..." until "Stop Camera" is clicked or an acknowledgement is received.

16. **App Behavior & UI Adjustments [Core]:** Update both apps to reflect the new command-based workflow:

17. **Android UI/UX:** If the Android app had a manual start/stop for sensors, you can now tie it to remote commands. Or you might choose to run headless (no UI) and purely respond to PC. But for safety, perhaps include a small UI indicating status (e.g., a TextView showing "Streaming On/Off", "Camera Recording..."). This helps if the phone is monitored or for debugging. Also perhaps a button to manually stop if needed (in case PC disconnects).
18. **Python UI:** Add elements for remote control:
    - Buttons: "Start Stream", "Stop Stream", "Start Camera", "Stop Camera", "Mark Event". For multi-device (in next phase) you might have per-device controls or a device selector dropdown.
    - When clicked, call the `CommandSender.send_command` with appropriate Command message.
    - Maybe reflect state: e.g., after sending "Start Stream", change button to disabled or switch to "Stop Stream" etc., assuming one stream at a time. This state could also be updated if the phone sends back info (we haven't explicitly provided a return channel except the data streams themselves; you could consider the phone sending a confirmation via an LSL event stream or even reuse the command stream to send an ACK from phone to PC).

19. **Acknowledgments (Optional):** Implement a simple ACK from phone after executing a command:

    - This could be as simple as logging or updating the data stream (e.g., push a special sample or event marker). Alternatively, phone could send a Command message back with a type like `COMMAND_ACK` (you'd add that to the enum).
    - Given this is a solo setup, a visual confirmation (like seeing the camera on) might suffice as feedback.

20. **Failure Handling for Commands [Core]:** Because network communication can fail, build in some basic reliability:

21. LSL's reliability: LSL uses TCP under the hood, so command messages sent will be reliably delivered in order as long as a connection exists [12]. If the phone app is not running or not yet listening, the PC's LSL outlet will just be waiting – once the phone joins, it should receive the latest commands (but any sent before phone connected would be missed). To handle this:
    - The PC could gray out controls until a phone stream is detected (so you only send commands when a device is present).

    - Alternatively, implement a handshake: e.g., phone when ready could send a "READY" command to PC (or simply the PC knows stream is up from Phase 2).
22. If a command is sent but no action occurs (due to some failure), have a timeout or retry mechanism. For example, if PC sends START_CAMERA but doesn't see any camera preview or any marker that it started within, say, 5 seconds, it could alert "No response from device – please check phone." In practice, one might just rely on user observation.
23. Ensure that STOP commands always attempt to execute even if some state flags say "not streaming" etc., to allow graceful recovery from mismatches (e.g., PC thinks phone is streaming but phone had stopped – an extra STOP won't hurt).
24. Security: not a big concern on a closed network, but note that LSL streams are visible to any LSL client on the network. If confidentiality or unintended control is a concern, you might incorporate stream authentication or unique IDs (out of scope for now but mention if relevant to lab).

Testing & Acceptance Criteria:

- **Unit Test – Command Message Creation:** On PC, test that when a user triggers each command, the correct `Command` proto is constructed. You can simulate button clicks in a test or call the handler functions directly. Verify fields: e.g., for Start Camera, `cmd.type == START_CAMERA` and `target_device` is set properly. Similarly on Android, test that given a `Command` object, the `handleCommand` executes the expected branch (could use dependency injection or mocking for sub-calls like `startCameraRecording` to verify it's invoked).
- **Integration Test – Command Round Trip:** This is best done in a controlled environment:
- Start the phone app in a test harness that listens for commands (you might automate this with an instrumentation test or a special test mode that logs received commands).
- On the PC side, send a test command (e.g., MARK_EVENT with a specific payload "TEST123").
- Verify the phone received it (maybe the phone app writes to a log or file upon receiving). This could be checked via logcat or a callback. In absence of automation, a manual check of logs is okay.
- This ensures the LSL command channel is functional end-to-end.
- **Manual Test – Camera Trigger:**
- Start PC and phone apps, ensure they connect. Ensure a sensor stream is running or ready.
- In the PC UI, click "Start Camera". The phone's camera should immediately begin recording (you might hear the shutter sound or see a recording indicator on phone). If a preview stream is implemented, the PC should display the incoming frames (verify the imagery corresponds to the phone's camera view).
- Let it record a few seconds, then click "Mark Event" on PC at a noticeable moment (e.g., wave hand in front of camera when pressing it).
- Click "Stop Camera" on PC. The phone should stop recording. Check that the video file is saved on the phone (perhaps using Android Device Monitor or a file explorer).
- Later, verify the video file's timestamp or content: The moment you waved (and pressed Mark) should be in the video, and the mark event timestamp logged on PC can be used to pinpoint that frame. This checks synchronization conceptually.
- Also test START/STOP of the sensor stream via PC if you implemented those: e.g., start/stop Shimmer sampling remotely.
- **UI/UX Test:** Ensure the UI properly reflects the states:
- e.g., If camera is recording, maybe the PC "Start Camera" becomes "Stop Camera" (toggle), and phone might show a red dot or text "Recording...". Stopping resets states.
- Ensure no inappropriate simultaneous commands (like if already recording, hitting start again shouldn't break anything – maybe disable the button or handle gracefully).

- **Error Handling Test:** Simulate the PC sending a command when phone is not ready (e.g., send START_CAMERA before phone app is launched). That command might just go nowhere; ensure the PC doesn't crash (our code wouldn't, it would still push to outlet which has no listeners). When the phone eventually connects, it will not retroactively get that command. This is acceptable, but document that the PC operator should start the phone app first or have a "connect" step.

Acceptance Criteria: The PC can successfully control the mobile app's behavior in near real-time. Specifically, the tester can initiate and stop sensor streaming and video recording from the PC interface. The phone responds promptly to each command (typically within <100 ms network delay). The camera recording on the phone is initiated remotely and the video is saved with correct timing. Any command that is sent is executed at most once (no duplicate actions), and the system can handle a sequence of commands (e.g., start stream -> start camera -> mark events -> stop camera -> stop stream) without getting out-of-sync. Timestamping is in place: for instance, when a camera start command is issued, the phone could log the LSL time it began recording, and the PC knows its send time – these can be compared to confirm synchronization (within a small delta considering network delay).

Also, by now, both **data streams** (GSR, etc.) and **event streams** (commands) are functioning. The design is extensible: adding a new command type later would involve adding to the proto and handling it in both places, following the same pattern.

Phase 4: Synchronization Enhancements & Fail-safe Mechanisms

High-Level Goals: Improve the system's robustness with respect to time synchronization and fault tolerance. While LSL already provides good time sync and reliability, this phase addresses scenarios like network outages, device crashes, or clock drift. We also incorporate any additional time-sync measures (e.g., NTP or RTCP) if needed and ensure the system fails gracefully (no data loss if possible, or at least data saved locally if connection is lost).

Development Subtasks:

1. **Time Synchronization Verification [Core]:** Although LSL's built-in clock sync is usually sufficient [9], perform a careful analysis and enhancement of timing:
2. Ensure all data samples have a timestamp at **source** (phone) using the same clock reference. We did this by using `LSL.local_clock()` on phone for GSR and marking events with LSL time. Double-check that for **camera**, we have at least an alignment point (e.g., log a message "Camera started at t=X" where X is `LSL.local_clock()`).
3. If feasible, synchronize the phone's system clock with the PC (e.g., via NTP). This is not strictly needed for LSL (which uses its own offset measurements), but having clocks roughly in sync can help when examining file timestamps or if any out-of-band data is used. Document a step in the README: "Ensure all devices are time-synced via internet or a time server before experiments" (common best practice).
4. If using any video analysis tools that rely on actual wall-clock time, you might consider embedding system time into video (like an overlay or metadata). But since LSL gives relative sync, it's optional.
5. Evaluate if **RTCP/NTP** sync is needed: For example, if at some point you stream video via RTP, RTCP could sync it. In our design, we are not using RTP, so we can skip. LSL's internal NTP-like sync covers multiple streams across devices automatically [13].

6. As a test, you might run a known sync-check: e.g., generate a synthetic event on phone and PC at the "same time" (like have phone and PC each log an LSL timestamp when something happens

like a manual button push simultaneously) and compare – they should differ by the measured offset only.

7. **Data Stream Failure Handling [Core]:** Augment the system to handle interruptions in streaming:

8. **Phone-side buffering:** If the PC or network goes down mid-session, the phone should not lose data. Implement a fallback on the Android app to **cache data locally** if the LSL outlet has no active inlets:
    - LSL `StreamOutlet` doesn't directly tell if an inlet is connected (it's mostly blind broadcast). But you can monitor an outlet's `have_consumers()` method (if available in liblsl Java) or simply detect if `push_sample` throws any error (usually it won't, it buffers).
    - Simpler: continuously record sensor data to a local file on the phone (like a parallel CSV or binary log) whenever streaming. Then even if PC disconnects, the phone still has the data. This is essentially a backup recording. Use timestamping as well so it can be merged later.
    - For video, we already save on phone, so that's fine.
9. **Automatic Reconnection:** On the PC side, if an inlet `pull_sample` starts timing out (meaning stream stopped or network lost), implement a reconnection attempt:
    - Use `pylsl.resolve_byprop` again in a loop every few seconds to see if the stream reappears. Or catch an exception from `pull_sample` and trigger re-resolve.
    - If the Android app crashed and restarted, its new stream might have a new ID. Perhaps base reconnection on stream **name/type** rather than exact unique ID, so you can find it again.
    - Communicate to user: show status "Reconnecting..." if connection lost.
10. **Graceful Shutdown:** Ensure that when PC stops recording or closes, it sends a STOP command to phone so phone can close resources (sensors, camera) safely. And if phone app is closed, it should close streams which will signal PC's inlet (inlet pull will error or return no data). Make sure to handle that on PC side (e.g., when `pull_sample` returns None for a while, conclude stream ended and update UI).

11. LSL has some built-in failure recovery for crashes (it can buffer data and recover) [14], but that typically requires the same stream to reconnect. Since our use-case is simpler, our manual reconnection covers it.

12. **System Health Monitoring (Optional):** Implement simple monitoring:

13. On PC, perhaps track the data rate of incoming samples and if it drops to zero unexpectedly, alert the user.
14. On phone, if sensor reading fails (e.g., Shimmer battery dies or BLE disconnects), send a command or at least log an error to PC (maybe via an LSL marker or by stopping the stream which PC will notice).

15. These touches make the system more user-friendly in a lab setting to quickly troubleshoot issues.

16. **Testing Sync and Failures:**

17. Create a scenario to test **time sync** accuracy: You could use an oscilloscope or a fast signal if it were analog, but with digital only, one idea is to toggle something on phone and PC at known times and compare logs. For example, flash the phone screen when an event marker is received, while PC also notes that time; using a high-speed camera to capture both might be extreme.

Given time constraints, rely on LSL's guarantees and maybe test with multiple devices (in Phase 5) for relative sync.

18. **Network loss test:** Start a session, then disable WiFi on phone for 10 seconds and re-enable:
    - Check that the phone continued recording offline during those 10s (the PC will have a gap; later you could merge from phone's backup).
    - PC should recover connection when network is back (assuming phone kept running LSL, it might automatically reconnect or you may need to re-discover).
    - At least ensure no crashes occur from such an event.
19. **Crash recovery test:** Simulate PC crash (close the PC app abruptly) while phone is streaming:
    - Phone should ideally detect loss (maybe not directly, but it will keep pushing buffered by LSL). Once PC app restarts and resolves stream, if the phone never stopped, LSL's **failure recovery** might allow PC to get buffered data or at least continue from current. LSL can buffer samples at sender for some amount if no consumer [15].
    - Alternatively, design phone to stop if no consumer for a long time to save battery, but only do so if absolutely needed.
20. **Timestamp consistency test:** If you recorded a backup file on phone and a file on PC (or just PC log), compare the timeline of events. For example, when PC went offline, the phone's local timestamps should continue smoothly. If you align phone and PC logs up to the disconnect point (using an event marker), the post-reconnection data should align again after accounting for the gap. This ensures your sync approach and reconnection logic work without time resets.

Acceptance Criteria: By the end of Phase 4, the system should demonstrate resilience and tight synchronization: - In normal operation, all data streams (sensor, commands, etc.) remain time-aligned within ~1–2 ms across devices, validated by inspecting timestamps (the built-in LSL sync achieving sub-millisecond precision on a LAN [9]). - If connectivity is lost or one side is restarted, the system recovers gracefully: **no crashes**, and data either continues seamlessly or is stored for later retrieval. For example, if the PC app is restarted mid-session, the user can re-attach to the ongoing phone stream and continue. - Users are informed of any disruptions (e.g., a status message "Reconnecting..." appears, rather than silent failure). - The phone stores a local copy of critical data (sensor CSV and video file) ensuring nothing is permanently lost due to network issues. - No uncontrolled backlog of data: buffers are appropriately sized so that if PC is offline for, say, 30 seconds, the phone doesn't run out of memory (either drop old samples or stop if needed with a warning).

This phase ensures the system is reliable for real experimental use, where things don't always go perfectly. It aligns with best practices: **timestamp at source** for every modality, and having a fallback for data recording (like local storage) in case of network failure, so the integrity of the experiment data is maintained.

Phase 5: Multi-Device Support & Scalability

High-Level Goals: Extend the architecture and implementation to support **multiple Android devices** streaming data simultaneously to a single PC controller. Allow independent or synchronized control of multiple phones, and provide the PC user with the ability to monitor and manage each device's data stream (and possibly camera) separately. Ensure the system scales to at least 2-3 devices without performance issues and that data from all devices remains synchronized.

Development Subtasks:

1. **Unique Stream Identification [Core]:** Modify the streaming setup to uniquely identify each device's streams:

2. Already in Phase 2, we used `source_id = deviceId` (or could use each phone's Bluetooth name or a configurable ID). Ensure each phone uses a distinct `device_id` (perhaps the user can set a name in the app, or use the phone model/serial).
3. The LSL `name` for the stream could remain "GSR" for all, which is fine because source_id differentiates. Alternatively, include the device name in the stream name (e.g., "GSR_AlicePhone"). Using meta-data is cleaner, so stick with `source_id` or add a custom field in the stream info meta-data like `<device>AlicePhone</device>`.

4. Similarly, for command streams: if using one command outlet for all, ensure the `Command.target_device` field is set for directed commands. If using separate command streams per device, give them distinct names or IDs (like "GSRCommands_Alice"). The simpler approach will be a single command stream where each phone checks the target field.

5. **Managing Multiple Inlets on PC [Core]:** Refactor the PC data receiver to handle multiple concurrent streams:

6. Instead of a single `GsrInlet`, create a manager class e.g. `DeviceManager` that keeps track of devices and their inlets:

```
class DeviceManager:
    def __init__(self):
        self.devices = {}  # dict of device_id -> DeviceData
    def discover_devices(self):
        streams = pylsl.resolve_byprop("type", "GSR", timeout=2)  # get
all GSR streams
        for st in streams:
            dev_id = st.source_id()
            if dev_id not in self.devices:
                inlet = pylsl.StreamInlet(st)
                self.devices[dev_id] = DeviceData(id=dev_id,
inlet=inlet, latest_value=None)
        # Remove or mark devices that disappeared (if any)
    def pull_samples(self):
        for dev in self.devices.values():
            sample, ts = dev.inlet.pull_sample(0)  # non-blocking
            if sample:
                dev.latest_value = sample[0]
                dev.latest_timestamp = ts
    # ... possibly methods to get data or stop a device
```

    - `DeviceData` could be a small dataclass holding info per device.
    - The UI can periodically call `discover_devices()` (or better, have a dedicated discovery thread or use LSL's continuous resolver) to catch new devices.
    - After discovery, continuously call `pull_samples()` in a loop or timer to update each device's data.
7. **UI representation:** If multiple devices, perhaps display each in a separate panel or tab:
    - For example, have a list of connected devices. Selecting one shows its real-time graph. Or show multiple graphs side by side if space allows.
    - For camera, you likely won't show multiple video previews simultaneously (too heavy), but maybe allow switching which device's camera feed to view.

    - Ensure each device's data is labeled (e.g., "Device Alice – GSR: 4.2 µS").

8. The command sender on PC should specify `target_device` when sending:

    - Provide UI controls to either select a specific device for a command or a "broadcast to all" option.
    - For synchronized start, you might allow multi-select devices and then "Start All" which sends separate START commands to each with their respective target (or one command with wildcard target that all phones obey).
    - If using one stream for commands, simply send multiple messages (one per device) for a group action, since LSL will deliver them quickly but not truly simultaneously (though within a few milliseconds which is fine). Alternatively, a broadcast command with target `"*"` (you decide that in your handleCommand, if `target_device == "*"`, any phone can respond).
    - Implement the logic in `handleCommand` on Android to either execute if `target_device` matches the phone's ID or if target is blank/"*".

9. **Scaling Performance Considerations [Core]:**

10. **Bandwidth:** GSR data is tiny, even multiple streams are fine. If all phones also stream preview images, be cautious: two phones sending 1 fps images doubles the bandwidth. Perhaps allow only one active preview at a time (e.g., only retrieve images from the currently selected device in UI).
11. **Threading:** Reading multiple inlets on PC – pylsl can handle it in one thread by round-robin polling, but you might use separate threads for each device to ensure one slow stream doesn't block others. Alternatively, pylsl has an async pull (but not in Python, it's blocking). A simple method: in `pull_samples()`, use a nonzero timeout like 0.0 to be non-blocking.
12. **UI updates**: If you show multiple graphs, updating many widgets frequently could slow the GUI. Optimize by limiting refresh rate (e.g., redraw at 20 Hz at most).
13. **Android side**: Two phones doing work – they are independent so that's fine; just ensure each can handle camera + sensor simultaneously (most modern phones can, but test with your devices).

14. **PC side**: The PC needs to handle multiple incoming streams. This is typically fine; LSL and modern PCs can easily handle dozens of streams (each just a few samples per second). The main load is any video processing.

15. **Synchronized Control Feature (Optional):**

16. Implement a "Global Start/Stop" that triggers all selected devices at once:
    - E.g., a "Start All" button that sends out start commands to each known device quickly. They won't be exactly simultaneous, but within network latency differences (~tens of ms). LSL's time sync means each phone will timestamp when it starts streaming, so you can align data post-hoc.
    - If more precision is needed, consider a scheduled start: e.g., send a command to each phone with a `timestamp` field in the future (like "START_STREAM at T=12345.67"). The phones could wait until their local `LSL.local_clock()` reaches that time to actually start. This is more complex but an interesting enhancement for perfect synchronization. However, network delay and clock offset errors (which are small) come into play. Given sub-ms sync, you could achieve start times within a few ms difference this way.
    - Test this if implemented: have two phones, use scheduled start, then compare the start timestamps in their data – they should be very close.

17. Provide a way to mark events across all streams: If PC sends MARK_EVENT with wildcard, all phones could log it (but that might duplicate the marker). Alternatively, PC itself can log the event and you treat it globally.

18. For multi-camera scenarios, synchronized start might help align videos.

19. **Testing & Acceptance Criteria for Multi-Device:**

20. **Multi-Device Connection Test:** Launch two (or more) Android apps (on different devices or an emulator if needed) and the PC app. Ensure the PC discovers multiple GSR streams. The UI should list both. Verify you can individually select each and see distinct data (maybe simulate different signals – e.g., one sensor on one person and another on another, or just use one actual sensor and a simulated one for test).
21. **Independent Control Test:** Start and stop streaming on Device A via PC, without affecting Device B:
    - PC sends START_STREAM to A -> A starts streaming (PC sees data). Device B remains off (no data).
    - PC then sends START_STREAM to B -> B starts streaming as well (now PC sees both).
    - PC sends STOP_STREAM to A -> A stops (PC no longer receives A's data, but B continues).
    - This tests that commands are properly directed.
22. **Simultaneous Control Test:** Use a group action:
    - E.g., select both devices and hit "Start All Streams". Both should start near-simultaneously. Check that both data plots begin updating.
    - Do the same for stop or camera if applicable.
23. **Multi-Camera Test (Optional):** If two phones can record video, try starting both cameras (maybe staggered to reduce load, or simultaneously if confident). Ensure both record and save their videos. This could be heavy, so consider it if needed.
24. **Synchronization Validation:** With two data streams, verify they are time-synced:
    - One method: if you have two GSR sensors on two people, you can have them do a simultaneous action (like both touch sensor at the same time on a cue) and see if the reaction is recorded at the same time. Or if you only have one sensor, you could simulate one stream by sending a duplicate data to two outlets for test – then their data should coincide.
    - Use the timestamps: if an event occurs (like a MARK_EVENT broadcast), see that each device's data timeline shows that marker at the same global time.
    - The Shimmer device might not be easily duplicated, so for testing, you could simulate a second stream by running a dummy app that sends a known pattern (e.g., a sine wave). Then verify PC gets it and aligns with first stream's timeline.
25. **Resource Usage Test:** Run the system with multiple devices for an extended period (say 30 minutes with 2 phones):
    - Monitor PC CPU and memory (should remain reasonable; plotting might be the biggest CPU hog, but still manageable).
    - Check phone battery usage – streaming data and recording video can be intensive. Ensure that in documentation you mention to keep phones charged or plug them in during use (practical tip).
    - No memory leaks: PC memory should not climb unbounded (if you keep historical data for plotting, consider a cap or downsample for long sessions).

Acceptance Criteria: The system successfully integrates multiple Android devices: - The PC can concurrently handle data from N phones (where N could be 2-3 as required by the project scope) and issue commands to each without confusion. The devices are identified clearly in the UI, and the user can select one or broadcast to all as needed. - Data from all devices is **synchronized**. For example, if two

devices are started nearly together, their data when plotted on a common timeline on the PC should align correctly (taking into account any intentional delays). - There is no crosstalk: Device A receiving a command meant for Device B should ignore it (our targeting ensures that). - The architecture supports extension: adding a new phone simply requires starting another app instance; the PC will pick it up. No hard-coded limit exists beyond perhaps UI practicalities. - The system remains stable under multi-device load. If one device disconnects, it should not break the others (e.g., if Device A goes offline, PC should handle that event – perhaps remove it from list and maybe alert "Device A disconnected"). - Multi-device support confirms that the design using LSL for synchronization is effective: as noted in prior research, LSL allows synchronization across multiple devices and data streams on the same network [11], which we have leveraged in our implementation.

At this stage, the technical core of the project is complete: a scalable, multi-device, synchronized data acquisition and control system.

Phase 6: Packaging, Deployment & Documentation

High-Level Goals: Prepare the applications for deployment in a real lab setting and hand-off. This includes building release versions (signed APK for Android, an executable for Windows), verifying installation on fresh systems, and writing comprehensive documentation (README or user manual) for setup and usage. Also, finalize any polishing tasks (UI improvements, bug fixes from testing) to ensure the system is maintainable and extensible for future work.

Development Subtasks:

1. **Android APK Release Build [Core]:**
2. **Signing Config:** In `android-app/app/build.gradle`, set up a release signing configuration. Either generate a new keystore (e.g., using Android Studio's Generate Signed Bundle/APK wizard) or use an existing one:

```
android {
    ...
    signingConfigs {
        release {
            storeFile file("keystore/fyp-keystore.jks")
            storePassword "yourpassword"
            keyAlias "fypgsr"
            keyPassword "yourkeypassword"
        }
    }
    buildTypes {
        release {
            signingConfig signingConfigs.release
            minifyEnabled false  // if you want, can keep false to
simplify debugging
            proguardFiles getDefaultProguardFile('proguard-
android.txt'), 'proguard-rules.pro'
        }
    }
}
```

3. Ensure the keystore file is kept safe (not committed if sensitive). If this is a thesis prototype and not public, security is less concern, but still follow best practices.
4. Build the release APK: via Android Studio ("Generate Signed APK") or command line `./gradlew assembleRelease`. Verify that the output `app-release.apk` is signed (you can use `apksigner` to check).
5. **Testing the APK:** Install the release APK on a test device (one that doesn't have the development environment). Make sure it runs without needing any dev extras. Specifically, check that the LSL native libs are included:
    - If you used the AAR, the `.so` libraries for all target ABIs should be packaged. Try on a device with a different architecture (if available) to be sure (e.g., arm64 vs armv7).
    - Also check that permissions (camera, Bluetooth) are properly requested in release build (they should be, same as debug).

6. Optimize the app icon, name, version code, etc., as needed for a professional touch. (Optional: set applicationId, versionName in Gradle, etc.)

7. **Windows Executable Packaging [Core]:**

8. Use **PyInstaller** to freeze the Python application into an executable. PyInstaller works well with PySide6 apps [16] :
    - Create a PyInstaller spec file or use a one-liner. For example, a one-liner:

      ```
      poetry run pyinstaller --noconsole --onefile -n "FYP-GSR-Controller" pc-app/fypgsr/main.py
      ```

      This will produce `dist/FYP-GSR-Controller.exe`.
    - `--noconsole` hides the console (since it's a GUI app).
    - `--onefile` packages everything into one EXE (which unpacks at runtime). One-file is convenient but can have slower startup; one-folder mode is fine too if easier.
    - `-n` sets the name of the exe.
    - Alternatively, write a `fypgsr.spec` to fine-tune:
    - Include any data files (e.g., icons or Qt UI files if you have them).
    - Ensure the pylsl library is included. **Important:** `pylsl` typically comes with a dynamic library (e.g., `liblsl64.dll`). PyInstaller might not automatically include it. You may need to add a hook or in spec file:

      ```
      a = Analysis([...],
                   binaries=[ ( 'path/to/Lib/site-packages/pylsl/liblsl64.dll', '.' ) ],
                   ...
      )
      ```

      Or use `--add-binary` in the CLI.
    - Test by running the EXE on your dev machine: it should launch the GUI and connect to devices like when run via Python.
    - Follow PyInstaller best practices (update to latest PyInstaller, etc. as needed [17]).

9. Once EXE is built, test on a **clean Windows PC** (one that doesn't have Python or dependencies). It should run standalone. Particularly, verify:
    - The PySide6 GUI shows up (meaning Qt plugins are included, which PyInstaller usually handles).
    - The LSL functions work – e.g., try connecting to a phone stream. If liblsl DLL was missing, it will error on resolve; if included properly, it should work.
    - If any issues, adjust PyInstaller settings (there are community hooks if needed, e.g., ensure `PySide6/plugins` are packaged).
10. (Optional) Create an installer for user convenience:
    - You could use something like InstallForge or Inno Setup to wrap the EXE into an installer that adds a Start Menu shortcut, etc. [18] . Given a lab setting, this might be overkill; simply providing the EXE or a zip with the EXE and a README might suffice.

11. Also ensure you have the necessary Visual C++ runtime included or available (PyInstaller usually includes everything needed, but note in docs if a VC++ redistributable is needed).

12. **Comprehensive README and Deployment Guide [Core]:**

13. Write a detailed README.md (or a separate documentation PDF) covering:
    - **Overview:** Purpose of the system, high-level architecture (maybe a diagram of PC and phones, and data flows).
    - **Installation (Android):** How to install the APK on an Android phone (enable "Install from unknown sources" if not on Play Store, etc.), any specific phone hardware requirements (Android version, Bluetooth, camera resolution).
    - **Installation (PC):** How to set up the PC controller:
    - If providing the EXE: "Download and run the .exe". Mention if Windows might SmartScreen flag it (since it's unsigned code); instruct user to trust it.
    - If running from source: Steps to install Python, Poetry, `poetry install`, etc. (This might be for developers, but good to include).
    - Note any firewall permissions: LSL uses multicast UDP for discovery, which might be blocked. Advise user to allow the app through the firewall if needed so that streams can be discovered on the network [19] (though on localhost or within LAN it typically works, but corporate networks might block multicast).
    - **Usage Workflow:**
    - How to start an experiment: e.g., "1) Launch the PC Controller app. 2) Turn on the Shimmer sensor and Android phone, launch the mobile app. 3) Wait for connection... etc."
    - Explain the UI controls: what each button does (Start Stream, Start Camera, Mark Event, etc.), and what indicators mean (e.g., device list, data graphs).
    - If multi-phone: how to add a new device (just launch another app instance; it will appear in PC GUI).
    - How to end and collect data: e.g., after stopping streams, the data might be automatically saved in PC or user needs to click "Save". If not implemented auto-save, instruct user to use LabRecorder or manual save screenshot etc. (You may have implemented simple logging to CSV; mention where that file is saved).
    - For video: tell user where to retrieve the video file on the phone (path on device storage) and how to match it with sensor data (e.g., using the timestamps or the fact that they started simultaneously).
    - **Troubleshooting:** List common issues and solutions:
    - PC cannot find stream: ensure phone and PC are on same WiFi/network, firewall off, etc.
    - LSL stream found but no data: maybe sensor not connected or not started.

- Time sync concerns: mention that system time differences don't matter due to LSL design, but to avoid confusion they can sync clocks via internet.
- If the PC app crashes or phone app crashes, how to recover (e.g., just reopen and it should reconnect; any partial data is saved on phone).
- **Configuration:** If there are configurable parameters (sampling rate, what sensors to enable, etc.), document how to change those (maybe via a config file or UI options).
- **Development notes:** (For future maintainers) how the project is structured, how to regenerate proto if message formats change, etc. Encourage adherence to the monorepo structure and how to run tests.

14. Also consider creating a short **Quick Start guide** for the lab (like a 1-page summary or a YouTube video demo if appropriate).

15. **Final Testing – End-to-End User Acceptance:**

16. Conduct a mock experimental session as a final acceptance test:
    - Set up two phones and the PC as if in a real scenario. Start everything, record 1-2 minutes of data + video, use markers at known events.
    - Stop and collect all outputs: the PC's recorded sensor data, and the phones' video files + any backup sensor file.
    - Analyze if everything aligns: e.g., plot GSR data and maybe watch video to see if peaks correspond to events visible in video given the markers.
    - If any discrepancy or bug is found (e.g., marker didn't log, or one phone's data stopped early), fix it now.
17. Test installation on a **different PC** (say a lab computer) by following your README instructions exactly. This ensures the documentation is accurate and the installer works.
18. Likewise, test on at least one more phone model to ensure broad compatibility (especially if using CameraX, test different camera hardware; if using Shimmer via Bluetooth, test pairing process on a fresh phone).

19. Perform a **performance sanity check**: on a lower-end PC or with more devices to see limits. This is optional but good to know (e.g., maybe beyond 5 phones the UI lags, etc., but if only up to 3 are needed, we're fine).

20. **Cleanup and Maintainability [Core]:**

21. Refactor any code that got messy during implementation. Ensure proper comments and docstrings, especially in tricky parts like LSL usage or proto message handling.
22. Perhaps set up a basic continuous integration (CI) pipeline (optional, if time): e.g., GitHub Actions to run Python unit tests, or a Gradle task for lint. This helps future maintenance but is optional for thesis unless required.
23. Make sure the project uses explicit versioning for releases. Tag the repo with a version (v1.0 for the delivered thesis artifact).
24. If this is open-source or for academic sharing, consider adding a license, and ensure no sensitive info (like keystore passwords) are in the repo.
25. **Extensibility:** In the documentation or comments, note how one would add a new sensor or feature. For example, "to add ECG streaming, define a new proto message and follow similar steps as GSR".

Testing & Acceptance Criteria:

- **Deployment Test:** A non-developer (e.g., a lab assistant or fellow student) can follow the README to install and run the system on their own. They should be able to replicate a simple recording without developer intervention. If they struggle at any step, refine the documentation.
- **Installation Integrity:** Verify the Android APK can be installed on any phone without needing debugging tools. Verify the Windows EXE runs on a PC without Python installed. This might include testing on Windows 10 and 11, etc.
- **Security/Permissions:** Check that all permissions are properly handled:
- On first run, the Android app should prompt for Camera and any other required permission (Bluetooth, Storage if saving video). Denying and allowing should be handled (document that user must grant these).
- The Windows app might trigger firewall—document to allow it for private network.
- **Final Functionality Regression:** Ensure all features still work in release build:
- Multi-device, commands, etc., as tested before, now in the packaged form. Sometimes release builds or frozen apps behave slightly differently (e.g., timing or performance). Do a quick run through all major features with the packaged apps.
- **Performance Benchmarks:** It's useful to note: e.g., measure end-to-end latency from sensor to PC display in the final setup (maybe ~50ms, you can estimate by the marker method or a quick reaction test). Also measure battery usage: e.g., 30 min of streaming + recording uses X% battery on phone (so user knows to charge if longer sessions).
- **Thesis Requirements Check:** Cross-check that all requirements of the project/thesis are met. For example, if the thesis expected a certain data quality or a user interface element, ensure it's there.

Once everything passes, you can consider the system ready. Summarize in your documentation how the design follows best practices: - Raw data recorded at source (yes, video on phone, plus optional local sensor log). - Ability to toggle streams (yes, via remote commands). - All events timestamped at the source and synchronized via LSL's protocol (which is based on NTP for clock offset [13]). - Multi-device support (yes, tested and functional).

Finally, prepare a short **maintenance guide** as part of docs: e.g., how to update the proto if new messages are needed, how to rebuild the Android app if moving to a new Android SDK, etc., to aid whoever continues this project (even if that's yourself later).

Acceptance Criteria: The project is deliverable to end-users: - The Android app is in a state that can be installed and used on experiment phones without connecting to Android Studio. - The PC app can be distributed as an .exe (or via source with clear instructions) and run on lab machines. - The documentation enables users to set up and troubleshoot the system. The system's features align with research needs, and any researcher using it can trust the data synchronization and have guidance on operating it. - The system is maintainable: a future developer (or the same developer later) can understand the code structure (helped by the modular monorepo and comments) and extend it (e.g., add a new sensor type, or adapt to a new phone) with minimal friction.

With Phase 6, the project is wrapped up as a professional, extensible tool ready for real-world use in experiments, fulfilling the master's thesis objectives.

Sources:

- LSL Android build & integration guidelines [5] [7]
- IntelliJ and Gradle protobuf configuration (enable Gradle build delegation for codegen) [4]

- Lab Streaming Layer documentation on time sync (NTP-based) and reliability [9] [14]
- Example of multi-device LSL usage in research (synchronizing multiple data streams across devices) [11]
- Packaging PySide6 apps with PyInstaller (compatibility with Python 3.6+ and PySide6) [16]

---

[1] How to use Java and python in intellij on the same project - Stack Overflow
https://stackoverflow.com/questions/53625164/how-to-use-java-and-python-in-intellij-on-the-same-project

[2] [3] [4] GitHub - google/protobuf-gradle-plugin: Protobuf Plugin for Gradle
https://github.com/google/protobuf-gradle-plugin

[5] [6] [7] Building for Android — Labstreaminglayer 1.13 documentation
https://labstreaminglayer.readthedocs.io/dev/build_android.html

[8] [13] Time Synchronization — Labstreaminglayer 1.13 documentation
https://labstreaminglayer.readthedocs.io/info/time_synchronization.html

[9] [10] [12] [14] [15] [19] GitHub - chkothe/labstreaminglayer: Multi-modal time-synched data transmission over local network
https://github.com/chkothe/labstreaminglayer

[11] GitHub - pmavros/shimmer_lsl_streamer: A simple script (and GUI) to stream data from a shimmer GSR device to LSL Lab Recorder, using the LabStreamingLayer.
https://github.com/pmavros/shimmer_lsl_streamer

[16] [17] [18] Packaging PySide6 applications for Windows with PyInstaller & InstallForge
https://www.pythonguis.com/tutorials/packaging-pyside6-applications-windows-pyinstaller-installforge/