# Milestone 2.6: Network Communication Client (JSON Socket)

**Objective:** Implement a network communication client on the Android side that interfaces with the PC controller using JSON-based messages over a TCP socket (or WebSocket). In this milestone, the PC will act as a server listening for connections, and each Android phone will act as a client that connects to it. This will allow the PC to send control commands to the phones and receive data from them, enabling synchronized multi-device operation under central PC control.

## Socket Connection Setup (Android Client)

* **Connection Type:** We will use a reliable stream socket over TCP. A plain TCP socket on a known port (e.g. port **9000**) is straightforward for our needs. (Alternatively, a WebSocket could be used for convenience since OkHttp supports it, but a raw TCP socket is simple and has no HTTP overhead.)
* **Initiating the Connection:** On app startup (or when the user taps a "Connect" button in the app UI), the Android app should spawn a background thread dedicated to network communication. This thread will attempt to create a java.net.Socket connection to the PC's IP address on the predefined port. The server IP could be configured manually by the user (entered in settings) or discovered via LAN discovery (e.g. mDNS) if time permits. Initially, we’ll implement manual configuration for simplicity.
* **Reconnection Logic:** The client should handle connection failures gracefully. If the socket fails to connect (e.g. PC server not available) or if it disconnects unexpectedly, the client thread should **retry periodically** (for example, wait 5 seconds and try again). This ensures that if the PC app comes online later or if there’s a temporary network glitch, the phone will automatically reconnect. The reconnection attempts should continue until a connection is established (or until the user explicitly disconnects/cancels).
* **Threading:** All network operations (connect, read, write) will happen on a separate thread (or via an AsyncTask/WorkManager) to avoid blocking the UI. We must not perform socket I/O on the Android main thread. The thread will continuously listen for incoming messages from the server and handle them (see *Handling Commands* below). Similarly, sending data from the phone (like status updates or frames) will be enqueued to this thread.

## JSON Message Protocol

All communication between the phone and PC will use **JSON-formatted messages** sent as text. Each message will be a standalone JSON object that includes a "type" field to indicate the message kind, along with additional fields carrying the relevant data. This simple schema makes it easy to parse and extend messages. We will define a set of message types for both directions (phone-to-PC and PC-to-phone):

### Phone-to-PC Message Types

These messages are sent **from the Android client to the PC** to inform the server of events or data. Examples include:

* **hello** – *Device Introduction*: Sent once upon connecting to the server. It identifies the device and its capabilities. For example:
* {"type": "hello", "device\_id": "Phone1", "capabilities": ["rgb\_video", "thermal", "shimmer"]}
* This tells the PC controller which device just connected (e.g., *Phone1*) and what data streams or sensors it provides. In this example, the phone can send RGB video, thermal camera data, and *Shimmer* sensor data. The PC can use this info to manage devices and ensure all required capabilities are available.
* **preview\_frame** – *Live Preview Frame*: Contains a single frame from a camera preview (useful if the PC wants live video feedback from the phone’s camera). The message might include which camera, a timestamp, and the image encoded as a base64 string. For example:
* {"type": "preview\_frame", "cam": "rgb", "timestamp": 123456789, "image": "<base64\_data>"}
* Here "cam": "rgb" indicates this frame is from the RGB camera (as opposed to the thermal camera), and "image" carries the image data. The PC might display this for monitoring or aiming the camera. **Note:** Large binary data like images are base64-encoded to keep the message JSON-friendly.
* **sensor\_data** – *Sensor Reading Update*: Contains readings from sensors (e.g., IMU data or external sensor like Shimmer). For example:
* {"type": "sensor\_data", "timestamp": 123456790, "values": {"accel\_x": 0.01, "accel\_y": -0.02, ...}}
* This could be sent at regular intervals or on change. The "values" object would include sensor readings. If using a Shimmer sensor, this might include multiple channels of data. This allows the PC to record or analyze live sensor streams in sync with video.
* **status** – *Device Status Update*: Contains health/status info about the device. For example:
* {"type": "status", "battery": 95, "storage": "10GB\_free", "temperature": 32.0}
* This might be sent periodically (say every minute) or on significant changes. It informs the PC of things like battery level (95%), available storage, device temperature, or any other relevant status. This helps the PC warn the user if a phone’s battery is low or storage is nearly full during a session.
* **ack** – *Acknowledgment/Response*: Sent in response to certain commands from the PC (see **Acknowledgements** below). This isn't spontaneously sent like the above, but when the phone needs to confirm or report the result of a PC command. For example: {"type": "ack", "cmd": "start\_record", "status": "ok"} would confirm that the start\_record command was received and executed successfully.

*(The phone could send other message types as needed, but these cover the core communication. Additional types might include error notifications, recording completion notices, etc., which can be added to the protocol later.)*

### PC-to-Phone Command Types

These messages are sent **from the PC controller to the Android client** to control its behavior. The Android app must be prepared to receive and handle these commands in real-time:

* **start\_record** – *Begin Data Recording*: Instructs the phone to start recording data for a session. It may include parameters such as a session identifier and flags for which data streams to record. Example command:
* {"type": "start\_record", "session\_id": "XYZ123", "record\_video": true, "record\_thermal": true, "record\_shimmer": false}
* This tells the phone to start a recording session labeled "XYZ123", and to record both RGB and thermal video (with record\_video and record\_thermal true) but not to record Shimmer sensor data in this session. Upon receiving this, the Android client should initiate all necessary recording services (camera recordings, sensor logging, etc., as implemented in earlier milestones).
* **stop\_record** – *Stop Data Recording*: Instructs the phone to stop the current recording session. Example:
* {"type": "stop\_record"}
* This implies that whatever was started (videos, sensors) should be stopped gracefully. The phone will then finalize the recorded files (and possibly prepare to send them back or save them for later retrieval). There could be an optional session\_id if multiple sessions are handled, but if the app only records one session at a time, it's not strictly necessary. After stopping, the phone might send an acknowledgment and possibly a message indicating where the data was saved or how to retrieve it.
* **capture\_calibration** – *Capture Calibration Image*: Requests the phone to take a special high-resolution photo (or a set of images) for calibration purposes. For example:
* {"type": "capture\_calibration"}
* Upon this command, the Android app should capture a still image from the RGB camera (and potentially one from the thermal camera if needed for calibration between the two). The behavior might be to save these images in a known location (or even immediately send them back to the PC as a follow-up message, though sending large images live could be optional). This command ensures all devices capture a calibration frame (like of a checkerboard or reference object) in sync.
* **set\_stimulus\_time** – *Synchronize/Set Time Marker*: This command carries a timestamp or time value that the phone should note, likely to synchronize events or stimuli across devices. For example:
* {"type": "set\_stimulus\_time", "time": 123456789}
* The meaning could be: “At UNIX time 123456789 (or a relative time), a stimulus occurs or should occur.” The phone might use this to align its data timestamps or trigger a specific action at that time (depending on the experiment design). Essentially, it helps in synchronizing all devices to a common timeline or marking an important moment.

*(Additional commands can be defined as needed. For instance, there might be future commands to change settings (like camera exposure, sensor sample rate) or to request a status update on-demand. The protocol is extensible via the "type" field.)*

## Handling Incoming Commands on Android

On the Android side, the network client thread will continuously listen for JSON messages from the PC. When a message arrives, the app needs to **parse the JSON** and **execute the corresponding action** promptly. Key implementation points include:

* **JSON Parsing:** We can use a library like **Gson** or **Moshi** to parse the incoming JSON string into a simple Java object/structure, or even use JSONObject from the Android SDK. Given the messages have a "type" field, one approach is to first parse just the "type" (e.g., by examining the raw string or using a lightweight JSON parser) to decide how to handle it. Alternatively, define a base Message class with subclasses for each type and use a JSON library to directly deserialize into the appropriate class. For simplicity, manual parsing with a switch on the "type" string is acceptable here due to the limited number of message types.
* **Executing Commands:** Based on the "type", perform the appropriate action in the app:
* **Start Record:** If a "start\_record" command is received, ensure the app is not already recording, then initiate the recording process. This may involve starting camera recording (via our RecordingService or equivalent from earlier milestones for both RGB and thermal cameras) and starting sensor logging (e.g., begin streaming data from the Shimmer or device sensors). Use the parameters provided (like session\_id for file naming or tagging, and flags to know which streams to record). After starting, the app might send back an acknowledgement (see next section).
* **Stop Record:** On a "stop\_record", stop all ongoing recordings. This involves stopping camera recordings (closing video files), stopping sensor captures, and finalizing any data files. Once everything is safely saved, the app can optionally start transferring the files to the PC (if that’s part of the design) or simply mark them ready. An acknowledgement or a follow-up message could inform the PC that the stop is complete and data is ready.
* **Capture Calibration:** For "capture\_calibration", trigger the device’s cameras to take a picture. This might require interacting with the camera APIs (which could be a bit different from the video recording flow). For example, if using Android's Camera2 API or a camera library, we would issue a capture request for a still image. We might store the calibration image locally (with a known filename) and/or send it directly to the PC (perhaps as a special message containing the image base64). If immediate transfer is too slow, the PC could later fetch it or the user could retrieve it, but at minimum the phone should capture and save it. Ensure both RGB and thermal images are captured in close succession for calibration (depending on project needs).
* **Set Stimulus Time:** On "set\_stimulus\_time", the app should record that timestamp. If the command implies an action at that time (like flashing something on screen or starting a sensor), the app should schedule that. However, it might simply be a marker; in that case, the app could sync its internal clock or note an offset so that all recorded data can be aligned to that reference. If needed, update any internal timing or schedule tasks accordingly.
* **Threading and Context:** Some actions (like camera operations) must run on specific threads or have certain timing. The network thread should offload heavy work to appropriate threads:
* For example, if interacting with the UI or camera hardware, consider using a Handler or sending an Intent/broadcast to trigger those actions in the main service or activity. We might integrate with existing services (e.g., if a RecordingService exists, it could expose methods or broadcasts for start/stop recording that the network thread can call).
* Ensure that the app remains responsive. Long operations (like capturing a high-res image or starting a recording) might need confirmation that they started, but they can run asynchronously.
* **Invalid States:** If a command is received that is not valid in the current state (e.g., a second "start\_record" while already recording), the app should handle it gracefully. This could mean ignoring the duplicate command, or responding with an error ack (explained below). The logic on the phone should track if it is currently recording or busy with a task, and avoid performing the same action twice or out-of-order.

## Acknowledgements and Response Messages

To maintain synchronization and inform the PC of the outcome of its commands, the Android client will send back **acknowledgment messages** for each significant command:

* After executing a command (or if execution fails), the phone sends a JSON message confirming the result. The format can be:
* {"type": "ack", "cmd": "<command\_type>", "status": "ok"}
* for success, or
* {"type": "ack", "cmd": "<command\_type>", "status": "error", "message": "<error\_details>"}
* if something went wrong.
* **Success Ack (status: "ok"):** Indicates the command was received and executed. For example, after a "start\_record", sending {"type": "ack", "cmd": "start\_record", "status": "ok"} tells the PC “the phone has started recording as instructed.” This gives the PC confidence to proceed (e.g., it might update the UI to show that Phone1 is recording).
* **Error Ack:** If the phone couldn’t comply (maybe an error occurred, or the command was inappropriate), it should inform the PC. For instance, if the PC sends "start\_record" but the phone’s cameras are not available or it’s already recording, the phone might respond: {"type": "ack", "cmd": "start\_record", "status": "error", "message": "Already recording"}. The message field provides a human-readable or coded reason. The PC can then alert the user or handle it (e.g., try stopping first or just log the failure).
* **Command Execution Latency:** Sending an immediate ack upon *receipt* of the command is useful to confirm delivery, but sometimes we might want to send the ack after the action completes. For example, we could send an ack as soon as we start processing a "stop\_record" (to confirm we got it), and maybe another message when the recording actually stops and files are finalized (e.g., a "recording\_stopped" event). For now, a single ack after action completion is fine for simplicity. If an action will take long, we might send a quick ack (“received”) then later a specific event (“done”). This design can evolve based on how the PC expects to manage state.
* **Keep-Alive/Ping:** Although not explicitly mentioned, in a long-running connection the server or client might occasionally need to know if the other side is still alive. We could repurpose the status message as a periodic heartbeat from phone to PC. Similarly, the PC could have a ping message type. Since we have frequent data (previews, status, etc.), a dedicated heartbeat may not be necessary unless the channel is idle for long periods.

By implementing acknowledgements, we ensure the PC and phones stay in sync about what commands have been handled, reducing uncertainty in the system.

## Robustness and Error Handling

Building a robust communication layer is crucial since network conditions can be unpredictable. Here are measures to ensure reliability:

* **Socket Connection Loss:** The app should detect if the socket is closed or encounters an error. If a read/write throws an IOException (e.g., connection reset, host unreachable), the network thread should catch it. In such cases, the app should mark itself as "disconnected" and then attempt to reconnect automatically (as described in the Connection Setup section). A short delay before reconnect attempts is wise to avoid tight loops if the network is down. This auto-reconnect behavior ensures the phone can recover connection without user intervention whenever the PC becomes available again.
* **Timeouts:** When attempting to connect or waiting for data, using timeouts can prevent the app from hanging indefinitely. For instance, Socket.connect() can take a timeout, and reading from the stream could use InputStream with a timeout or use a non-blocking approach. If the app doesn't hear anything from the server for a long time (and isn't sending anything either), it might decide to reconnect as well. However, since the PC is mainly sending commands and presumably listening continuously, lack of communication might not always indicate a problem (it could just mean no commands needed). Still, implementing a periodic ping or using the OS socket keep-alive options could help detect broken connections (e.g., if Wi-Fi drops but socket doesn't realize immediately).
* **JSON Integrity:** Because we are streaming JSON messages, we have to ensure each message is parsed completely and correctly:
* If for some reason a partial message is received or the JSON is malformed, the client should handle the parsing exception. Possibly skip or discard that message (and maybe request a resend if the protocol supported it). Logging such incidents would help debugging.
* Using a clear delimiter or length (discussed in the next section) will help us avoid mixing messages or partial reads. The code should accumulate incoming bytes until a full JSON message is detected, then parse it.
* **Thread Safety:** When the network thread receives a command and needs to affect other parts of the app (like starting a service or changing a setting), proper synchronization or thread-safe communication is needed. For example, the network thread might use a handler or send a broadcast/intent to the main thread or a service to actually start a recording (since starting camera preview/record might need to run on the main thread or a camera-dedicated thread). We must ensure these cross-thread calls are safe and don’t cause race conditions (for instance, don’t allow two start commands to be processed concurrently).
* **State Management:** Keep track of the app’s state (idle, recording, etc.) to validate incoming commands. This prevents, say, two simultaneous recordings. If a command is inappropriate, decide whether to ignore it or send an error ack. The design of the PC controller could be such that it won’t send logical mistakes (like double start), but our client should not assume perfect input. It should guard against unexpected sequences gracefully.
* **Error Logging and Alerts:** If something fails (network error, JSON parse error, action error), log it (to file or logcat) for debugging. Optionally, update the UI to inform the user (e.g., "Connection lost, retrying..."). While the system is mostly autonomous, having visible status (like a connection indicator in the app) can help users during testing.

By considering these edge cases, we aim to make the communication channel robust, so that the overall system remains stable even in less-than-ideal network conditions.

## Message Framing and Data Transmission

Because we are sending JSON over a raw TCP stream, we need a strategy to **frame** messages so that the receiver can tell where one message ends and the next begins. JSON text itself doesn’t have a built-in delimiter (concatenating two JSON objects yields an invalid JSON, since the parser wouldn’t know where to split). We will implement a framing protocol to solve this:

* **Newline-Delimited JSON:** One simple approach is to send each JSON message followed by a newline character \n. The receiving side can read until it encounters a newline, treat that chunk as one JSON string, parse it, then continue reading for the next message. This works because in our usage we will not include unescaped newline characters inside the JSON data. (By JSON specification, newline in strings must be escaped as \n, so raw newline byte will only appear as our delimiter.) This technique is commonly known as *newline-delimited JSON* or *NDJSON*, and it leverages the fact that JSON formatters typically don't include literal newlines in the output[[1]](https://en.wikipedia.org/wiki/JSON_streaming#:~:text=Streaming%20makes%20use%20of%20the,be%20used%20as%20a%20delimiter). It’s easy to implement and to debug (you can even test by sending messages via telnet or netcat line by line).
* **Length-Prefixed Messages:** Another robust method is to send a **length header** before each JSON payload. For example, we could send a 4-byte integer (in network byte order) that specifies the length of the upcoming JSON string (in bytes). The receiver first reads 4 bytes to get the length N, then reads the next N bytes which correspond to one full JSON message. This approach allows any content (even binary) since we rely on length rather than special characters. It handles cases where JSON might inadvertently contain the chosen delimiter character. The downside is that it’s a bit more involved to implement parsing, and not human-readable on the wire. However, since we might send large binary data (images) as base64, using a length prefix ensures we safely get the entire base64 string even if it contains characters like = or others that could confuse a naive delimiter.
* **Choice of Framing:** We will likely implement **length-prefixed framing** for maximum safety and clarity in parsing. This means on the Android side, before sending a JSON message, we will: (1) convert it to a UTF-8 byte array, (2) get its length, and (3) first send a 4-byte representation of that length, followed by the bytes of the JSON. On the PC side, the server will mirror this logic: read 4 bytes to get length, then read that many bytes for the JSON string. This approach is common in many protocols and avoids any ambiguity. (If we were to use newline-delimited JSON, we must ensure the JSON encoder does not insert newlines. We could manage that by not pretty-printing JSON and by using Base64 encoders without line breaks for image data. Both approaches are valid; in fact, newline-delimited JSON works fine since JSON strings cannot contain raw newlines[[1]](https://en.wikipedia.org/wiki/JSON_streaming#:~:text=Streaming%20makes%20use%20of%20the,be%20used%20as%20a%20delimiter). But for learning and explicitness, we'll go with length prefix.)
* **Binary Data Consideration:** We are encoding images as base64 specifically to keep the messaging simple and text-based. This increases size (~33% overhead) but ensures the image can travel as JSON. Alternatively, we could send binary data outside JSON (especially if we used a binary-friendly protocol or WebSockets), but that adds complexity. With base64 in JSON, framing is even more important to separate messages. In the future, if performance becomes an issue, we might consider sending raw binary frames (WebSocket can send binary messages easily, or a custom binary protocol), but that's beyond this milestone.
* **Flushing and Buffering:** We should also be mindful to flush the output stream after sending a message to ensure it actually goes out promptly (especially if using buffered I/O). The InputStream reading on the other side will block until data is available, so sending and flushing each message helps keep things real-time.

By implementing a clear framing strategy (length-prefixed in our case), we guarantee that both the phone and PC can exchange JSON messages without confusion, even when messages are streamed back-to-back.

## Future Considerations and Alternatives

While the above covers the core implementation, a few additional notes and possible future improvements:

* **Using WebSockets:** If we opted for WebSocket instead of a raw TCP socket, much of the framing and reconnection logic could be handled by existing libraries. For example, **OkHttp’s WebSocket** support can automatically handle ping/pong heartbeats and binary vs text frames, and works on a background thread for sending/receiving[[2]](https://developer.squareup.com/blog/web-sockets-now-shipping-in-okhttp-3-5/#:~:text=Unlike%20the%20traditional%20request%2Fresponse%20model,native%20support%20for%20web%20sockets)[[3]](https://developer.squareup.com/blog/web-sockets-now-shipping-in-okhttp-3-5/#:~:text=Enqueue%20text%20or%20binary%20messages,even%20Android%E2%80%99s%20main%20thread). WebSockets also easily traverse some networking environments due to being an HTTP upgrade. However, using a plain socket is perfectly fine for a controlled environment (all devices on a LAN). Given our timeline, we chose the simpler raw socket approach, but in the future switching to WebSocket might simplify some parts or allow using a higher-level protocol (like Socket.IO if we needed rooms, etc.).
* **Discovering the Server:** Currently, we assume the user will input the server IP or that it’s known. In a polished product, we might implement **mDNS (Multicast DNS)** or a simple UDP broadcast-based discovery so phones can find the PC automatically. For instance, the PC could broadcast its presence and IP on the local network, and phones could listen and auto-connect. This would improve usability (no manual IP entry). This is not critical for functionality and can be added later as an enhancement.
* **Security:** Our JSON socket protocol is currently unencrypted within the local network (which is acceptable in a closed LAN or lab environment). If this needed to be used in a wider network or internet, we’d need to consider encryption (e.g., use TLS sockets or wss for WebSocket) and authentication (so that only authorized clients connect to the server). We might also restrict commands to avoid any malicious usage. These are beyond the current scope but worth noting.
* **Performance:** JSON and base64 have overhead. For our use (likely small number of devices and moderate data rates), this is fine. If performance or bandwidth becomes an issue (say, sending high-frequency frames), we could optimize:
* Throttle preview\_frame rate or compress images more.
* Eventually consider a binary protocol for image frames (or use a separate channel/stream for bulk data).
* Use efficient JSON parsing (avoiding excessive object creation).
* But until profiling shows an issue, the clarity and ease of JSON is a good trade-off.
* **PC Server Implementation:** Although this milestone is focused on the Android client, for completeness: the PC side will have a server socket listening on port 9000, accepting connections from each phone. It will need to handle multiple clients (if more than one phone) and track them (like mapping device\_id from the "hello" message to that socket). It will broadcast or target commands to each client as needed. The PC should also implement similar robustness (handling disconnects, etc.). The protocol we designed is symmetrical enough that implementing it on the PC (likely in Python, C#, or Java) is straightforward using JSON parsing and socket libraries.

## Conclusion

By the end of Milestone 2.6, we will have a functioning **network communication layer** between the Android phones and the PC controller. Each phone, upon launching the app, will connect to the PC and announce itself. The PC can then send JSON-formatted commands like *start recording, stop, capture image,* etc., and the phones will execute those actions in unison. The phones will also stream back useful data (previews, sensor info, status) and acknowledgements. This achieves a key goal: turning the previously independent recording devices into a **synchronized, centrally-controlled system**.

With this in place, the groundwork is laid for the PC application to coordinate experiments or data collection across multiple devices seamlessly. The next steps after this would likely involve refining the user interface on both ends, testing the reliability over real Wi-Fi conditions, and making any adjustments to ensure timing alignment and data integrity across the system. Overall, Milestone 2.6 transforms the project from a set of isolated device apps into a **networked multi-device platform**.

[[1]](https://en.wikipedia.org/wiki/JSON_streaming#:~:text=Streaming%20makes%20use%20of%20the,be%20used%20as%20a%20delimiter) JSON streaming - Wikipedia

<https://en.wikipedia.org/wiki/JSON_streaming>

[[2]](https://developer.squareup.com/blog/web-sockets-now-shipping-in-okhttp-3-5/#:~:text=Unlike%20the%20traditional%20request%2Fresponse%20model,native%20support%20for%20web%20sockets) [[3]](https://developer.squareup.com/blog/web-sockets-now-shipping-in-okhttp-3-5/#:~:text=Enqueue%20text%20or%20binary%20messages,even%20Android%E2%80%99s%20main%20thread) Web Sockets now shipping in OkHttp 3.5!

<https://developer.squareup.com/blog/web-sockets-now-shipping-in-okhttp-3-5/>