# Milestone 3.6: File Transfer and Data Aggregation (Step-by-Step Guide)

## Goal and Overview

**Objective:** Automate the transfer of recorded data files from each phone to the PC at the end of a recording session. This ensures all videos and sensor logs are centralized on the PC for analysis, eliminating manual copying and reducing human error. After the operator stops recording (sending a "stop\_record" command to devices), the system will automatically collect all the resulting files (e.g. RGB video, thermal video, sensor CSV logs, etc.) from each phone and save them in the session folder on the PC.

**Key Challenges and Requirements:**  
- **Reliable Transfer Protocol:** Use the existing socket connection (over Wi-Fi/local network) to request and receive files. This avoids setting up separate FTP/HTTP servers and keeps the design simple.  
- **Chunked Data Transmission:** Large files (like high-resolution video) must be sent in chunks rather than one giant message, to avoid memory issues and allow possible recovery if interrupted.  
- **Data Integrity and Verification:** Ensure that files are fully received and intact (using known file sizes or checksums for verification). Consider the ability to resume transfer if a connection drops (so a partially transferred file doesn’t require restarting from zero).  
- **Multiple Devices:** If multiple phones are used, the PC should handle retrieving files from each device, ideally concurrently or sequentially without user intervention. The system must not overwhelm the network – transfers can be staggered or parallelized carefully.  
- **Minimal Impact on Recording Workflow:** The transfer should initiate *after* recording stops (since live 4K video streaming would be too heavy). The phone may need a moment to finalize files (flush video data to disk) before sending, so the PC should trigger transfer at the appropriate time (e.g. right after receiving confirmation that recording stopped).

By the end of this milestone, the data collection process will be fully automated: the moment a recording session ends, all data is pulled into the PC’s session folder structure. This streamlines the workflow for researchers.

## Protocol Design for File Transfer

We will extend the existing JSON message protocol between the PC and phone clients to support file transfer. Below is the proposed message sequence and format for requesting and sending a file:

1. **PC requests file from Phone:** After sending the stop command and confirming the phone has stopped recording, the PC sends a JSON message to the phone:

* {"type": "send\_file", "filepath": "<device\_file\_path>", "filetype": "<description>"}

1. filepath is the path or filename of the data file on the phone (e.g., "session123\_phone1\_rgb.mp4"). This can be predetermined by the PC if it knows the naming scheme (for example, using a session ID and device ID). Alternatively, the PC could send just a file type identifier and let the phone determine the exact file path.
2. filetype (optional) is a descriptor (e.g., "video" or "sensor\_log") primarily for logging or conditional handling. The crucial part is the filepath which tells the phone what to send.
3. **Phone acknowledges and provides file info:** Upon receiving the send\_file request, the phone prepares to send the file. It first responds with a JSON message containing file metadata:

* {"type": "file\_info", "name": "<filename>", "size": <filesize\_bytes>}

1. name is the name of the file (e.g., "video1.mp4" or "session123\_phone1\_rgb.mp4").
2. size is the total size in bytes. The PC will use this to know how much data to expect and to verify completeness.
3. This message indicates that the phone found the file and is about to start sending it. (If the phone cannot find the file, it should respond with an error message instead – e.g., {"type": "error", "message": "file not found"} – so the PC can handle it gracefully.)
4. **Phone sends file data in chunks:** After file\_info, the phone streams the file content as a sequence of chunk messages. To keep the protocol simple and still in JSON, the file bytes will be encoded in Base64 and sent in manageable blocks. Each chunk message might look like:

* {"type": "file\_chunk", "seq": 1, "data": "<base64\_string>"}

1. seq is the sequence number of the chunk (starting from 1, incrementing by 1 for each chunk). This helps with debugging or reordering if ever needed.
2. data is the Base64-encoded string of the binary chunk. We choose Base64 so that the binary data can be represented as text within JSON. **Note:** Base64 encoding will increase data size by ~33%[[1]](https://en.wikipedia.org/wiki/Base64#:~:text=Base64%20encoding%20causes%20an%20overhead,by%20the%20inserted%20line%20breaks), but on a local network this overhead is acceptable for simplicity. (No line breaks will be inserted in the Base64 data, to keep it continuous JSON; the 33% overhead is the main impact.)
3. **Chunk size:** We will send, for example, 64 KB of raw data per chunk (which becomes around 85 KB of Base64 text per message). This size is a balance between efficiency and memory usage. Sending in 64KB blocks means thousands of chunks for very large files, but avoids any single JSON message being extremely large. The chunk size can be tuned if needed (it could be larger on a robust network or smaller if memory is a concern). Using a consistent chunk size also simplifies the logic.

The phone will loop reading the file and sending chunks until the entire file is sent. Each chunk is sent as a separate JSON message over the socket.

1. **End-of-file marker (optional):** Once the phone finishes sending all chunks, it can send a final JSON message to mark completion, for example:

* {"type": "file\_end", "name": "<filename>"}
* This tells the PC that no more chunks will follow for that file. In practice, because the PC knows the expected size from file\_info and is counting bytes received, the file\_end message might be redundant – but it's a nice explicit confirmation. It also allows the phone to signal completion even if the exact byte count was slightly off or if we want to double-check names. The PC, upon receiving file\_end, can finalize the file save process.

1. **PC acknowledges receipt (optional):** After the PC has successfully received and reconstructed the file, it could send back a confirmation:

* {"type": "file\_received", "name": "<filename>", "status": "ok"}
* This lets the phone know the transfer was successful. If we plan to have the phone delete files after transfer to free space, this acknowledgement would be the trigger (the phone would wait for this before deleting the local copy). If an error occurred (file corrupted or incomplete), the PC could send a status "error" to possibly retry. This acknowledgement step is optional but recommended for robust operation.

**Alternate Approaches Considered:** We considered using a separate raw binary stream (or an HTTP/FTP transfer) to send the files, to avoid the overhead of Base64 and JSON parsing. For example, the PC could start a temporary file server, or the phone could push the file via HTTP POST. However, these approaches introduce more complexity (additional servers, different protocols, handling binary framing). By reusing the existing socket connection and JSON messaging, we keep implementation simpler at the cost of some overhead. In our design, the simplicity and not having to manage multiple connection types is worth the trade-off. If performance becomes an issue (due to the Base64 overhead or JSON parsing costs), we can revisit this decision in the future. For now, the JSON chunk approach should be sufficient on a local network and easier to implement in Python on both ends.

## Class and Module Breakdown

Implementing this feature will touch both the PC application and the phone client code. Below is a breakdown of how to incorporate the file transfer logic into the system’s architecture:

### PC-Side Components (Desktop Application)

* **SessionController / Orchestrator Module:** This high-level component (possibly the one that starts/stops sessions) will initiate file transfers once a session ends. It knows which phones are involved and what data to collect. It will iterate over each connected phone and trigger the transfer of expected files. This could be part of an existing SessionManager class. Key responsibilities:
* After issuing the "stop\_record" to all devices, monitor for confirmations from phones that recording has stopped or files are ready. (If the phone sends a “recording\_stopped” message or similar, use that as a signal that files are finalized.)
* For each device, call a method like requestDeviceFiles(device) to start pulling files from that phone.
* Manage the overall sequence or concurrency of transfers (discussed below in concurrency section).
* **DeviceClientHandler / ConnectionHandler (per phone):** If the PC app already uses a per-connection handler (e.g. one thread or async handler per phone socket), that class should be extended to handle new message types (file\_info, file\_chunk, file\_end, etc.) and to send send\_file commands. Possible additions:
* sendFileRequest(filepath) method: Constructs and sends the {"type": "send\_file", ...} message to the phone. Optionally include which file we want.
* State variables to manage incoming file data: e.g., expectedBytes (from file\_info), receivedBytes, current file being received, and an open file handle for writing.
* In the message-processing logic (where JSON messages from phones are handled), add cases:
  + On file\_info: Open a new file on PC for writing (in the designated session folder). Use the provided name or the original filepath to name the local file. Store expectedBytes = size from this message. Initialize receivedBytes = 0. Possibly allocate a buffer or ensure the directory exists.
  + On file\_chunk: Decode the Base64 string back into binary bytes, write those bytes to the open file. Append mode writing in chunks. Increase receivedBytes by the chunk length. (The phone’s seq can be logged or used to ensure ordering, but with TCP, messages should arrive in order. If one chunk is missed or out of order, something is wrong at the protocol level. Still, logging seq can help debug if needed.) Optionally, update a progress indicator (like print or GUI progress bar) using receivedBytes/expectedBytes.
  + On file\_end: Close the open file. Verify that receivedBytes equals expectedBytes (and perhaps check file size on disk as well). If everything matches, mark the transfer successful (and maybe send a file\_received ack). If there's a discrepancy, log an error and possibly take action (like request a re-send).
* **File assembly logic:** It might be helpful to encapsulate the file writing logic in a helper class or function, say FileReceiver. For example, when a file\_info is received, you could instantiate a FileReceiver object with the target path and expected size, and then feed it chunks as they arrive. This object can handle writing and tracking progress. If the design is simpler, you can just handle it within the handler class itself with some variables as described.
* **Session Folder Management:** Ensure that for each session (or each recording run) there is a dedicated folder on the PC where incoming files will be saved. The folder might be named by timestamp or session ID. The PC should create this folder when the session starts (or when stopping/collecting, if not already). The file paths for saving will be something like Session\_2025-07-28\_001/phone1\_rgb.mp4, etc. Managing this location might be part of a higher-level class, but it’s worth noting to have the directories ready before writing files.
* **Logging & Error Handling:** The PC side should log key events: when a file transfer starts, progress (perhaps in percentages), when it completes, and if any issues occur (like a missing file or a transfer interruption). If a transfer fails mid-way (e.g., socket disconnects), the PC should catch that. Potentially, it could attempt to reconnect to the phone and resume (more on resumption below). Any exception (like failure to open file, decode base64, or write) should be caught and logged so it doesn’t crash the entire app.
* **User Interface Feedback (if applicable):** If the PC app has a UI, consider showing the status of file transfers – e.g., a progress bar per file or at least a message like “Transferring video1.mp4 from Phone 1… (50% complete)”. This assures the user that the system is working and gives an idea of how long to wait if files are large. Since the file size is known from file\_info, a simple percentage can be calculated. In a console app, even printing “Received X of Y bytes” periodically is useful.

### Phone-Side Components (Mobile App)

* **Network Message Handler:** On the phone, there is presumably a socket listener or client thread that receives JSON messages from the PC (like start/stop commands). This handler needs to be updated to recognize the "send\_file" request. Typically, this might be in a loop parsing incoming JSON messages. Pseudocode inside phone app might be:
* msg = receive\_json()  
  if msg["type"] == "send\_file":  
   filepath = msg["filepath"]  
   handleSendFileRequest(filepath)  
  elif msg["type"] == "stop\_record":  
   ...
* You will implement handleSendFileRequest(filepath) to perform the file transfer. (If the phone app is Android in Java/Kotlin, the logic is similar: in the network listener, add a case for the "send\_file" command and call a method to send the file.)
* **File Transfer Sender:** The phone will need to open the requested file from storage and send it in chunks. Key considerations for the implementation:
* **File Access:** Ensure the app has permission to read the file. If the video and log files are saved in the app's private storage, you can open them directly. If they are in shared storage (e.g., DCIM or external storage), you might need READ permissions (and for devices running Android 10+, you might be using scoped storage or saving in app-specific directories). Make sure this is sorted out during development (in testing, adjust the path or permissions as needed).
* **Reading and Sending Loop:** Implement a loop to read the file and send chunks:
  1. First, get the file’s total size (and maybe derive a file name). This can be done via file API (e.g., File.length() in Java, or os.path.getsize in Python, etc.). Immediately send the file\_info JSON back to the PC with this size and name.
  2. Open the file in binary mode for reading.
  3. Set a chunk size (64 KB as decided, or a define constant).
  4. Read a chunk of up to that size from the file. If using Python on the phone, you can directly Base64 encode the bytes (using Python’s base64 module) and then send the JSON message. If using Java/Kotlin, you can use Android’s Base64 utility (android.util.Base64) to encode bytes to a Base64 string. Be mindful of memory: do this chunk by chunk rather than reading the whole file at once.
  5. Send each chunk as a JSON message with the format described (type: "file\_chunk", seq: n, data: "<base64>"). You might create the JSON string manually or use a library/JSON serializer. Ensure the message is sent fully before reading the next chunk (in Python socket, send() or sendall(), in Java, writing to output stream).
  6. Continue until end-of-file. After the loop, optionally send the file\_end message. Then close the file.
  7. It could be useful to pause briefly or yield between chunks to avoid flooding the network buffer, but if using TCP, it will naturally throttle if the PC can’t keep up. Still, the phone could insert a tiny sleep if needed (likely not needed on a robust Wi-Fi, but something to keep in mind if performance tuning).
* **Threading on Phone:** If the phone’s network handling is on a background thread already (common in network clients), that thread can perform the file reading and sending. Ensure this does **not** run on the main UI thread of the app, because reading a large file and sending data can take time and we don’t want to freeze the UI. If your current implementation uses a background service or thread for socket communication (which it should), you can reuse that. If not, you might spawn a new Thread or use an AsyncTask/Kotlin coroutine from the point of receiving the send\_file command. For example, upon "send\_file", start a new thread that executes the file sending loop so the socket listener remains responsive (especially important if we allow multiple file requests sequentially).
* **Progress (Optional on Phone):** The phone could log how much of the file it has sent, but since the PC is the one needing progress, it’s not strictly necessary. However, for debugging, you might want to log seq number or bytes sent so far, so if something stops you know where it left off.
* **Multiple Files Automation:** The phone might record multiple files in a session (e.g., video.mp4, thermal.mp4, sensors.csv). We have to ensure each of those can be transferred. We have two design options:
* **PC-driven requests:** The PC can send a send\_file request for each expected file one after the other. For example, PC sends for "video.mp4", phone sends it; then PC sends for "thermal.mp4", etc. This way the phone only ever handles one file at a time. This is simpler and avoids any multitasking on the phone. We will implement this method.
* **Phone-initiated list:** Alternatively, the phone could, after receiving the stop command, proactively send a message like {"type": "session\_files", "files": ["video.mp4","thermal.mp4","sensors.csv"]} indicating what it has ready. The PC could then loop through that list requesting each. This requires the phone to know which files to send – which it does, since it created them. This approach is also fine; it adds an extra step but can make the PC more dynamic (it doesn’t have to guess filenames).
* In our implementation, since the PC likely knows what was recorded (it instructed the phone what to do), we can go with PC-driven requests for known filenames (possibly constructed using a session ID or timestamp to avoid confusion). We will implement the PC to sequentially request each file type it expects. To keep it robust, we can incorporate a check: if a phone responds with an error or a file is missing, log it and continue with others.
* **Cleaning up after Send (Optional):** After successfully sending files, the phone could choose to delete or archive the files to save space, *but only* after it knows the PC got them (to avoid data loss). This is a future improvement. For now, it’s safe to leave files on the device and maybe have the researcher clean them periodically or implement a cleanup strategy later. We mention this so the system design anticipates that phone storage could fill up after many sessions; automatic cleanup after confirmed transfer is a logical next step.

## IDE Configuration and Setup

Setting up your development environment properly will help implement and test this milestone efficiently:

* **PC Application (Python) Setup:**
* Make sure the Python environment has any needed libraries. Likely we use the standard library (socket, json, base64, maybe threading). If the PC app is part of a larger project (e.g., using an IDE like PyCharm or VS Code), ensure that the project interpreter is correct and dependencies are installed. No special external libraries are needed for the basic file transfer, since base64 is in the standard lib.
* If using an IDE, you might want to configure logging output to be easily viewable. For example, in PyCharm, enable the console to show all stdout prints or use a logging framework to file. This will help in seeing the progress logs of file transfers.
* Organize the code: you might create a new Python module, e.g., file\_transfer.py, containing helper classes or functions like FileReceiver or any logic that doesn’t fit neatly in existing classes. This module can be imported into your main server code. Alternatively, if the project is small, just add the logic in the main script or existing classes. Just maintain clarity (comments and docstrings for new methods) since file transfer involves multiple steps.
* **Mobile App (Android) Setup:**
* Open the Android project in Android Studio (if the phone client is a native app). Ensure you have connectivity permissions in the AndroidManifest.xml (<uses-permission android:name="android.permission.INTERNET" />). This should already be present if the app communicates over sockets. If you plan to read/write files to external storage, also ensure <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" /> (for older Androids) or proper Storage Access Framework usage for newer Androids. However, if files are within app-internal storage, no additional permission is needed beyond Internet.
* Locate the part of the code that handles incoming socket messages. This might be in a background Service or a networking Thread class. Prepare to add a new case for "send\_file". If the code is well-structured, it might have a method like onMessageReceived(JSONObject msg) that you can modify. If it’s more ad-hoc, find where the JSON is parsed. In any case, plan where to insert the call to the file sending routine.
* If using Python on the phone (for example, a Python script running via QPython or similar), ensure the environment on the phone has the needed libraries (base64, etc.). If it’s an Android app using Chaquopy or Kivy, the approach is similar – just ensure you can open files and send data. (Given typical scenarios, we assume an Android Java/Kotlin app, so we focus on that.)
* **Concurrency considerations:** If the phone uses a single thread for network I/O, sending a large file could block receiving new commands. This is okay since we don’t expect new commands while transferring (the session is stopping). But if the PC tries to request multiple files in quick succession, ensure the phone can handle it: one approach is to queue the file requests and process sequentially. Alternatively, handle one send\_file at a time – perhaps the PC waits for one to complete (file\_end) before sending the next request, which is simpler. We will implement the PC that way (sequential per device). So the phone doesn’t necessarily need a queue, it will just respond to one request at a time.
* Increase socket buffer if needed: By default, sockets have an OS buffer (often a few KB to few tens of KB). We are sending large data, but in chunks. You might not need to change anything, but if performance is slow, consider using socket.setSendBufferSize / setReceiveBufferSize on either end to allow larger in-flight data. This is an optional tuning.
* **Synchronization of Stop and Send:** Ensure that the phone actually has finished writing the file to storage by the time we start sending. If the phone writes video data to a file on stopping, there might be a slight delay (a few milliseconds to seconds for flushing encoder data). A conservative approach: the PC could wait for a “stop confirmation” message from phone. For example, maybe the phone, after stopping recording, already sends something like {"type":"recording\_stopped","files": ["..."]}. If not, we can simulate a short delay or have the PC attempt the file transfer after sending stop and possibly waiting a second. In testing, watch out if the first chunk the phone sends appears corrupted or empty – that could indicate the file wasn’t fully closed yet. In that case, add a small delay or send a confirmation from the phone when it’s ready.
* **Directory Structure on PC:** As part of setup, decide where the session folder resides. For example, you might have a base directory like C:\ResearchSessions\ or ~/sessions/ and within it a folder per session. The PC code should know or be configured with this. Setting this up in a config file or at the top of the script is useful. Ensure the program creates the directory if it doesn’t exist. This is more of a one-time configuration, but it’s essential for saving files without error.

## Implementation Steps

Following is a step-by-step plan to implement the file transfer and aggregation feature. It’s divided into logical steps with testing checkpoints:

**1. Define New Message Types in Protocol:**  
- Decide on the JSON fields for send\_file, file\_info, file\_chunk, and file\_end as described above. Add these to any documentation of the protocol you maintain. If you have an enum or constants for message types in code (on both PC and phone), update them to include these new types.  
- **Checkpoint:** Write a short section in code comments or docs listing the new messages and their JSON structure. This will guide your implementation on both sides and ensure consistency.

**2. Implement Phone-Side File Send Handler:**  
- In the phone’s code, implement the function (or method) that will handle a "send\_file" request. For instance, handleSendFileRequest(filepath). Within this function: - Open the file (path may be absolute or relative to a known directory – ensure you construct it correctly if the request gives only a filename). If the file cannot be opened, send an error message back (and perhaps log the error on phone).  
- Determine file size and name, then send the file\_info JSON back to PC.  
- Loop to read the file in binary mode in chunks (e.g., 65536 bytes each). For each chunk: - Base64-encode the chunk bytes into a string. (In Python: base64.b64encode(chunk\_bytes).decode('ascii'). In Java: Base64.encodeToString(chunkBytes, Base64.DEFAULT) which will include newline by default – use Base64.NO\_WRAP to avoid newlines, so the JSON isn’t broken by line breaks.)  
- Create a JSON object/string with type:"file\_chunk", the sequence number, and the data string. Send it over the socket.  
- You might want to flush the output stream if using buffered output, though send or write typically sends immediately for TCP.  
- Increment sequence count and continue until EOF.  
- After the loop, send the file\_end message. Then close the file handle.  
- **Threading:** If needed, ensure this runs on a background thread. For now, implement it straightforwardly; we can refactor into a thread if we find it blocks other operations.  
- **Checkpoint:** Test this function in isolation if possible. For example, you could temporarily call handleSendFileRequest on the phone with a known small file and have the phone send to the PC (or a test server) to verify it sends the expected messages. If you don’t yet have the PC side ready, you can use a network debugging tool (or simple socket listener) to capture what the phone sends. This is tricky without the PC implemented, so you might skip directly to integrated testing. But at least log on the phone what it *would* send: e.g., print the first chunk’s size or sequence count, to verify the loop logic is correct.

**3. Implement PC-Side Receiving Logic:**  
- On the PC application, update the message handling to process incoming file\_info, file\_chunk, and file\_end messages from the phone:  
- When file\_info is received: create/open a file on disk for writing. Use the session folder and maybe subfolder for that device. For the filename, you might use the provided name or construct one (for example, prefix with device ID or session ID to avoid naming conflicts if multiple phones have a file with same name like video.mp4). Record the expected size. Initialize a counter for received bytes = 0.  
- When file\_chunk arrives: decode the data field from Base64 back to bytes. Append those bytes to the file (write in binary mode). Increase the received byte count. You can verify sequence (seq) if you want to ensure none were skipped (if a chunk is out-of-order or missing, it indicates a serious communication issue; with TCP this is unlikely unless a message was lost due to disconnect). In normal operation, seq is mostly for logging (e.g., “Received chunk 10 from phone 2”).  
- During chunk processing, consider memory: writing each chunk immediately to file and not storing them all in memory is important for large files. Your approach should stream the data to disk. Using Python’s file write in a loop is fine. Just ensure to open the file once and keep it open during the transfer (opening for each chunk would be slow).  
- Optionally, after writing each chunk, you can calculate how many percent of the file is done and log it. e.g., progress = (receivedBytes/expectedBytes)\*100. Only do this calculation periodically (maybe every 10 chunks or every few MB) to avoid too much logging. Or update a GUI progress bar if you have one.  
- When file\_end is received: close the file. Then check if receivedBytes == expectedBytes. If they match, the file is fully received. You might also trust the file\_end as the final marker. If there’s a mismatch (say the socket closed unexpectedly and we missed some chunks), handle that (perhaps log an error and mark file incomplete). If all good, you can send back a file\_received confirmation to the phone (if you implement that).  
- If the phone doesn’t send an explicit file\_end, you can infer completion when receivedBytes reaches expectedBytes. At that point, you can close the file and send ack. (Still, keep a timeout just in case – e.g., if no data arrives for a while and receivedBytes < expectedBytes, something went wrong. A timeout can trigger an error handler to abort or retry.)  
- Integrate this logic into the existing network reading loop on PC. Likely you have something like:

data = socket.recv( ... )  
message = json.loads(data)  
handle\_message(message, from\_device)

You might need to handle the case where data is very large (JSON string spanning multiple TCP packets). Ensure your socket reading code can assemble a full JSON message that might be ~85KB or more. If you use something like recv(1024) in a loop until a newline, consider increasing buffer size or reading until you detect end of JSON. Another tactic: since each JSON message is complete (and our chunks aren’t human-typed lines, they may not end with newline), it might be better to implement a protocol where each JSON message is prefixed with its length or separated clearly. If your current implementation already handles messages (perhaps by delineating by } or newline), test it with a large JSON to confirm it works.  
- **Checkpoint:** With both phone and PC code written for one file transfer, do a controlled test with a small file: - Perhaps create a dummy file of known content (e.g., a small text file or a small image file) on the phone. - Run the system: have the phone connect to the PC (as in a normal session), then manually trigger a file request. This could be done by simulating a "send\_file" command (maybe using a test function or after a short dummy recording). - Observe on PC if the file gets created and matches the original. You can verify by comparing file sizes and contents. If it’s a text file, check content integrity. For binary (image), perhaps compute an MD5 hash on both ends to ensure they match. This will validate your base64 encode/decode and no data corruption in transit. - If any issues arise (like JSON parse errors or mismatched data), debug now before proceeding to real recordings.

**4. Automate Multi-File Transfers per Device:**  
- Now that one file can be transferred, extend the PC logic to request all necessary files from each phone when a session ends. For example, if each phone produces 3 files per session, the PC should send three send\_file requests in sequence to that phone. This can be done in a simple loop:

for file in files\_to\_get:  
 send\_file\_request(phone, file)  
 wait\_until\_file\_received(phone, file)

The wait\_until\_file\_received could simply mean your code waits until the file transfer completes (you can track this via the logic in step 3, e.g., once you get file\_end for that file). This ensures you don’t bombard the phone with multiple file requests at once. It keeps things straightforward: the phone will finish sending one file, then handle the next request.  
- Determine files\_to\_get: This could be hardcoded types (if you know each phone always sends "rgb.mp4", "thermal.mp4", "sensors.csv"), or if you implemented the phone to send a list of files it produced, use that list. For now, you can hardcode the expected files by name pattern. Perhaps you stored the session ID and each phone’s ID; use those to form filenames. *Example:* If session ID is 101 and phone ID is A, you might have told the phone to name its video session101\_A\_rgb.mp4. So PC can construct that name and ask for it. For the sensor log, similarly session101\_A\_sensors.csv. (Make sure the naming scheme on phone matches what PC expects. It might be easier to have the PC *tell* the phone what name to use when starting recording, e.g., include the session ID in the start command. But if not, you can guess or request a list.)  
- Implement the loop for each phone connected. If your PC SessionController already has a list of active device connections, iterate through them. This could be done sequentially (first get all files from Phone 1, then Phone 2, etc.), or in parallel threads. More on concurrency in the next step. Initially, try sequentially to keep it simple and reliable.  
- **Checkpoint:** Simulate a scenario with two “phones” (maybe one real phone and one dummy client on another PC or emulator) if possible. Trigger a session stop and file retrieval. Verify that the PC successfully collects files from both. If you don’t have multiple devices handy, you can simulate by connecting the same phone twice or running a second instance of the phone client code (if it’s a script) that uses a different ID and provides dummy files. The main point is to ensure your loop correctly handles multiple connections. Watch the logs to see that it requested files from each and saved them distinctly.

**5. Handle Concurrent Transfers from Multiple Phones (if needed):**  
- Transferring files sequentially from each phone is simple, but if you have many devices or large files, this could take a long time. To speed it up, you can transfer from multiple phones in parallel. Since each phone has its own socket connection, doing this concurrently is feasible.  
- One approach is to spawn a new **thread** for each phone’s file-transfer sequence. For example, when stop is pressed, for each connected phone start a thread that performs the file requests (as in step 4) for that phone. This way, phone1 and phone2 can send simultaneously. Python’s threading is suitable here because the tasks are mainly I/O-bound (waiting for network and disk), so the GIL won’t be a big problem. Alternatively, if using an async framework, you could await on both transfers concurrently.  
- If implementing threads: ensure that your file writing on disk is thread-safe per file. If each thread writes to a different file, that’s fine. Use locks if any shared data structures (like if all threads report progress to a shared UI, guard that). Also be mindful of disk IO: writing two large files at once might contend on a slow disk, but on modern SSDs or if files are on different drives, it’s usually okay.  
- The phone side likely doesn’t need special changes for this, as each phone is independent. Just be sure the phone is only dealing with one file at a time for its own connection.  
- If network bandwidth is a concern (e.g., all phones on the same Wi-Fi router), parallel transfers might strain it. But if the router and PC can handle it, this is the fastest approach. If not, you could limit to, say, 2 phones at a time or stick to sequential. You can observe performance in testing and adjust.  
- **Checkpoint:** If possible, test with two phones simultaneously sending a moderately sized file. See if both come through correctly. Monitor the network (router stats or PC network usage) to confirm it’s handling it. Check that the files are not getting mixed up – our protocol has separate sockets so they shouldn’t, but ensure your code keeps data separate per connection (which it should if each has its own thread/handler). Also ensure the PC UI or logs remain responsive (if logging a lot from two threads, it might intermix lines, but that’s okay).  
- If any thread crashes or a phone disconnects mid-transfer, ensure that one thread’s exception doesn’t stop the others. Handle exceptions inside each thread and perhaps communicate back to main program if needed (e.g., mark that phone’s transfer failed, but let others continue).

**6. Implement (Optional) Resumption Support:**  
- A robust system might allow resuming a failed file transfer. This is an advanced step, but worth designing now if possible. If a phone loses connection mid-file (say Wi-Fi drops), the PC could reconnect and ask only for the remaining bytes instead of the whole file again. To do this: - PC would need to know how much it already received (easy, we track receivedBytes). - PC could send a new message to phone like: {"type": "send\_file", "filepath": "...", "offset": <bytes\_already\_received>} indicating start from a certain position. Or use {"type":"send\_file\_resume","filepath":..., "chunk\_index": X}.  
- The phone’s file handler then would seek to that byte offset in the file (e.g., using file input stream skip or Python file.seek) and start sending from there. It might need to adjust chunk sequencing (either continue the seq count or start over — seq is mainly informational anyway).  
- This requires the phone to support that parameter. It’s easier to implement now than retrofitting later. If you choose to, add logic: if the send\_file JSON has an "offset" field, use it. If offset is 0 or missing, start from beginning. This way the same message type can handle both cases.  
- On the PC side, you’d have to detect a failure: e.g., if socket disconnects unexpectedly, you could reopen connection (maybe the phone app automatically reconnects on disconnect) and then send the resume request. This is complex to test, so you might implement the basics (the command and the phone’s ability to seek) but not fully test it until a real scenario arises. At minimum, log that resumption is available.  
- If not implementing now, plan it for the future. For initial implementation, it might be acceptable that if a transfer fails, the user manually restarts the session or at least the transfer.

**7. Verification and Integrity Checks:**  
- After each file transfer completes, verify the integrity of the file: - **Size check:** Compare the file size on PC (via os.path.getsize for example) with the expected size from file\_info. They should match exactly. If not, something went wrong (log error).  
- **Playback/format check:** For videos, a quick check is to try opening the video (if you have a script that can verify MP4 integrity or just manually play it after the test). For sensor CSV, try opening or parsing a few lines to ensure it’s not truncated. These can be part of testing procedures.  
- **Checksum (optional):** For critical data, computing an MD5 or SHA-256 hash on the phone and on the PC and comparing would be the ultimate integrity test. This could be automated: phone could compute a hash of the file and send it either in file\_info or after sending all chunks. Python’s hashlib could compute MD5 quickly even for large files (maybe stream it while reading to avoid reading file twice). If you include this, PC on receiving can compute its own hash and compare. This is optional and might be overkill for local network, but it’s a nice guarantee if needed.  
- Log a message "File X received successfully, size Y bytes." If using a UI, show a checkmark or similar. This gives the operator confidence that the transfer worked.  
- **Checkpoint:** Run a full end-to-end test of a session: 1. Start a recording on at least one phone via the PC app as usual (even a short 5-second recording, to generate files). 2. Stop the session using the PC app. 3. Observe the file transfer automatically happening. If possible, have a console or log window open to see progress messages (or a UI element showing progress). 4. Wait for completion and then check the session folder on PC. Verify all expected files are present. 5. Open the video file on the PC (using VLC or any player) to ensure it's not corrupted and is the correct footage. Open the sensor log in a text editor or Excel to ensure data looks plausible. 6. If multiple devices were used, verify each device’s files are there (maybe named with device identifier to avoid confusion). 7. If anything failed (say one file didn’t transfer), check the logs for clues (did we get an error, or did the process hang?). Debug accordingly.  
- Repeat the test for different scenarios: a longer recording (to test larger file), multiple phones simultaneously, etc., to ensure robustness.

**8. Final Wrap-Up:**  
- Once tests pass, finalize the implementation by cleaning up any debug prints (or converting them to formal logs). Make sure to handle any edge cases discovered. For example, what if the phone had no thermal camera and thus no thermal video file – the PC might request a file that doesn’t exist; the phone should reply with an error and PC should handle it (skip and maybe warn). Ensure this doesn't crash the system.  
- Document the usage: update the user manual or internal docs to note that file transfer is automatic. From the user’s perspective, after they hit "Stop", they just need to wait until a message or indicator shows all files are collected. You might show a GUI message like "Session complete. All data saved to Session\_123 folder."  
- Keep in mind future maintenance: if the file naming conventions change or new data types are added (e.g., another sensor), you'll need to update the transfer requests accordingly. Try to make the code adaptable (maybe derive file names systematically or ask phones for list of files).

## Additional Considerations and Tips

* **Performance Tuning:** For large videos, the Base64 encoding/decoding is the biggest overhead in CPU and size. If performance is a concern (e.g., transferring a 2 GB file taking too long), consider increasing chunk size (maybe 256 KB chunks) to reduce the number of JSON messages. Also, ensure release builds of the app (on phone) are optimized – Java/Kotlin code in release mode should handle base64 fine. In Python, base64 encoding is C-optimized, so it's fairly fast. The network will likely be the bottleneck (e.g., a 2 GB file over Wi-Fi might take a few minutes).
* **Avoiding JSON for Data Chunks:** In the future, you could implement a hybrid mode: use JSON for control messages and then switch to a raw binary mode for file data. For example, after file\_info, the PC and phone could agree to send raw bytes until the expected size is met, then return to JSON. This avoids the 33% overhead. But it complicates the socket handling (you’d have to either temporarily suspend JSON parsing or have a separate socket). Since our aim is to minimize complexity, we didn't choose this route initially. But it's something to consider if throughput needs to be improved later.
* **Ensuring JSON Parser Robustness:** Large JSON messages (especially with Base64 content) can be a bit heavy to parse. If you run into memory issues or slow parsing on the PC side (less likely in Python for, say, tens of MB strings, but still), you could process the chunks without full JSON parsing by searching the received string for the "data" field, etc. However, that's usually unnecessary. If using Python’s json.loads, it should handle it if given the complete string. Just be careful if you accumulate data from recv in a loop – you need to join the parts correctly. Tools like socket.makefile() can help treat socket as a file and read line by line if your JSON messages are newline-terminated. Or you might use a library like json-stream for huge JSON. In testing, see how it performs.
* **Concurrent Sends from Phone (not needed now):** We decided phones will send one file at a time after each request. An alternative might be the phone automatically starting to push files once recording stops, without waiting for requests. That would shift control to phones. We chose the PC-request model for clarity and to avoid collisions (multiple phones pushing at same time can still be handled with threads, but at least each phone doesn’t overwhelm itself). This is just to note why we went with a pull model instead of push.
* **Deletion of Files on Phone:** As mentioned, consider a strategy for file cleanup. Perhaps after a successful transfer, the PC could send a command {"type": "delete\_file", "filepath": "..."} or a general {"type": "session\_cleanup"} to instruct the phone it’s safe to remove local copies. You may implement this in a later milestone. It's good to think about because phones have limited storage, and if doing many sessions, old data should be cleared. For now, researchers can manually clear or swap SD cards if needed, but an automated approach is ideal in the long run.
* **IDE Debugging:** Use your IDE’s debugging tools to step through the file transfer if you encounter issues. For example, you can put breakpoints on the phone after sending a few chunks to see if PC received them. Or vice versa, break on PC’s chunk handler to inspect the decoded bytes. Since this is networking, a lot of debugging will be via logs, but an interactive debugger can still help if both ends are running in controlled environments (for instance, you might run a phone client in an emulator or as a local Python process to step through it).
* **Testing Environment:** If possible, test with the phone and PC on the same Wi-Fi network router (typical use). Also test with them connected via a mobile hotspot or different network scenarios if that’s in scope, to ensure the transfers remain reliable. Given it’s TCP, it should be fine as long as the connection is stable.

## Conclusion

Milestone 3.6 establishes an automated pipeline for aggregating all recorded data to the PC immediately after each session. We designed a straightforward extension to the existing client-server protocol to request and transfer files using Base64-encoded chunks over the socket connection. We provided a detailed breakdown of the necessary code changes on both the PC and Android phone sides, covering how to handle the messages, read/write files in chunks, and maintain data integrity. We also outlined how to set up the development environment to implement these changes and thoroughly test them, including handling multiple devices and large files.

By following this guide, you should be able to implement the file transfer feature step-by-step, verifying each part as you go. Once completed, the system will significantly streamline the data collection workflow: as soon as a recording is stopped, all videos and sensor logs from each phone will automatically appear in the PC’s session folder, ready for analysis. This reduces manual effort and potential mistakes, and paves the way for further automation such as immediate data processing or backup. With Milestone 3.6 achieved, the platform becomes a more efficient and reliable multimodal data collection system.

[[1]](https://en.wikipedia.org/wiki/Base64#:~:text=Base64%20encoding%20causes%20an%20overhead,by%20the%20inserted%20line%20breaks) Base64 - Wikipedia

<https://en.wikipedia.org/wiki/Base64>