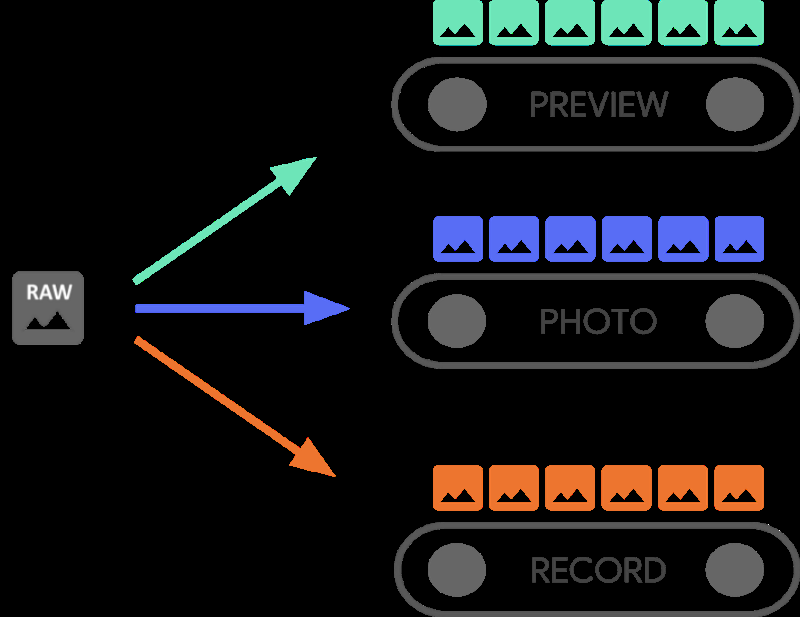
# Milestone 2.5: Live Preview Streaming Implementation

**Objective:** Enable live preview streaming from the Android phones to the PC controller application for real-time monitoring. In this milestone, each phone acts as a client that captures preview frames from its cameras, compresses them, and transmits them efficiently over the network to the PC. This allows the PC operator to see a live video preview (from the visible-light camera and/or thermal camera) during recording, helping verify framing, focus, and sensor status in real time.

## Preview Frame Capture

Modern Android Camera2 API supports using multiple output streams from the same camera sensor simultaneously[[1]](https://developer.android.com/media/camera/camera2/capture-sessions-requests#:~:text=A%20single%20Android,more%20than%20one%20stream%20simultaneously). This means we can capture a live preview for streaming in parallel with the ongoing high-resolution recording and any on-screen preview. **Figure 1** illustrates how one camera sensor’s raw feed can be split into parallel pipelines for different purposes (e.g., preview, photo, and recording)  
. We will leverage this capability by adding an extra low-resolution output for network streaming in addition to the existing outputs.

* **Using Camera2 Preview Streams:** We likely already set up a preview Surface (SurfaceView or TextureView) in Milestone 2.2 for on-screen display. To grab frames for streaming without disrupting the main recording, we will add an ImageReader as a second output target in the CameraCaptureSession. The ImageReader can be configured with a **lower resolution** (for example, 1280×720 or even VGA 640×480) and an appropriate image format (YUV or JPEG). This additional output will receive a copy of each camera frame at the chosen resolution, in parallel with the main preview and recording outputs. The Android camera device can handle multiple outputs if the resolutions and formats are within hardware capabilities[[1]](https://developer.android.com/media/camera/camera2/capture-sessions-requests#:~:text=A%20single%20Android,more%20than%20one%20stream%20simultaneously). Most modern devices support at least one extra low-res stream alongside a high-res recording stream.
* **Visible Light Camera:** For the RGB camera on the phone, we will configure an ImageReader with a down-scaled resolution (e.g. 854×480 or similar) to capture preview frames. This smaller size ensures the preview frames are lightweight to process and send. The format can be ImageFormat.JPEG to let the camera hardware encode each frame to JPEG, or ImageFormat.YUV\_420\_888 if we plan to manually convert to JPEG in software. Using a hardware JPEG stream is preferred to offload compression work to the camera ISP and get a ready-to-send JPEG byte array for each frame.
* **Thermal Camera:** For the infrared/thermal sensor, we likely obtain frames via the device’s SDK (for example, through Topdon SDK callbacks). These frames are typically low resolution (often around 160×120 or 320×240 for thermal imagers) and come as either grayscale bitmaps or byte arrays. We will reuse those incoming frames for preview streaming. Since the thermal frames are already being received (and displayed or recorded) via the SDK, no additional Android Camera2 setup is needed for them. We just need to capture the latest thermal image available whenever it’s time to stream a frame.

**Efficiency Consideration:** High-end phones can produce multiple outputs in parallel, but there are limits. If the extra preview stream’s resolution or frame rate is too high, the camera pipeline might start dropping frames[[2]](https://developer.android.com/media/camera/camera2/capture-sessions-requests#:~:text=Parallel%20processing%20suggests%20that%20there,frames%2C%20it%20starts%20dropping%20them) or reducing quality. By choosing a modest resolution (e.g., 480p) and a low frame rate for the network stream, we ensure the overhead is small. The preview frames for streaming will not be needed at full 30 fps – a slower rate (perhaps 2–5 fps) is sufficient for monitoring. This reduces CPU/GPU load and bandwidth usage.

## Frame Encoding (Compression)

Raw camera frames are large, so we must compress them before sending to keep network usage manageable. We will use image compression (preferably JPEG) for both the visible and thermal preview frames:

* **JPEG via Camera Hardware:** The simplest approach is to configure the ImageReader for the visible camera with ImageFormat.JPEG. In this mode, the camera’s hardware encoder will directly produce compressed JPEG images for each frame. This yields a byte[] for each preview frame that is already compressed and ready to send, avoiding heavy CPU usage on the phone for encoding. The JPEG quality and size can be tuned via Camera2 parameters if needed (though at small resolutions the default quality should be fine). Each 640×360 or 854×480 JPEG frame might be on the order of a few tens of kilobytes, which is reasonable for network transmission.
* **Software Compression (Alternative):** If using a YUV ImageReader (for more control or compatibility reasons), we would get YUV420 pixel data. We would then manually compress it to JPEG using a library or Android’s Bitmap.compress() function. However, this is less efficient and can introduce latency, so we prefer the hardware JPEG route when possible.
* **Thermal Frame Format:** Thermal camera frames, being grayscale and low-res, will be very small. We can convert the thermal frame (which might be a grayscale Bitmap or byte array) into a PNG or JPEG. PNG could make sense for grayscale since it’s lossless and the images are simple (and likely small in file size due to limited detail and color), but JPEG is also acceptable if easier – the size difference will be minor at such low resolutions. The key is to produce a compressed image buffer (PNG or JPEG) of each thermal frame for sending. Given the small size (possibly only a few kilobytes per frame), this won’t stress the network.

By compressing each frame to a binary image format, we drastically reduce the data that must be sent over the socket. For example, a raw 854×480 YUV frame is about 1.2 MB, but a JPEG of the same frame might be 50 KB – a 95% reduction. This makes real-time streaming feasible over Wi-Fi.

## Networking (Client Side Transmission)

With compressed frames in hand, the phone needs to send them to the PC in real time. We assume there is an existing network connection to the PC (likely a TCP socket) and a JSON-based message protocol in use (from earlier milestones). We will integrate the preview stream into this network layer. There are two approaches to send the image data over the socket:

* **Option A: Base64 within JSON:** Encode the image bytes as a Base64 string and put it in a JSON message. For example, the phone could send a JSON object like: {"type": "preview\_frame", "camera": "visible", "image": "<base64\_data>"}. This is very simple to implement since we can stick with the JSON text protocol – the image becomes just another field as a string. The downside is size overhead and encoding/decoding cost: Base64 encoding inflates the data by about 33% (every 3 bytes become 4 ASCII characters)[[3]](https://en.wikipedia.org/wiki/Base64#:~:text=Base64%20encoding%20causes%20an%20overhead,by%20the%20inserted%20line%20breaks), and converting to/from Base64 uses some CPU. For occasional low-res frames, this overhead is acceptable for simplicity. The PC, upon receiving the JSON, would decode the Base64 back into the image bytes and then display the frame.
* **Option B: Binary Protocol or Hybrid:** Send a small header or JSON message followed by the raw binary image bytes outside of JSON. For instance, send a JSON like {"type":"preview\_frame","camera":"visible","length":12345} first, then send the 12,345 bytes of JPEG data directly over the socket. This avoids the Base64 size overhead and encoding step. However, it complicates the protocol: we need to manage a clear boundary between JSON text and binary data in the stream. We’d also need the PC side to read the length and then read that exact number of binary bytes from the socket. It’s more complex to implement compared to keeping everything in one JSON string.

**Approach Choice:** We will likely start with **Base64-in-JSON** for simplicity and consistency with our existing JSON messaging system. It requires no changes to how messages are parsed on the PC side (just decoding the base64 string to get the image). The extra bandwidth usage (33% overhead) is a reasonable trade-off given our low frame rate and resolution. For example, if each JPEG frame is 50 KB, the Base64 string will be ~67 KB. At 2 frames per second that’s ~134 KB/s, which is about 1.1 Mbps – easily handled by Wi-Fi networks. If we later find this to be a bottleneck, we can consider switching to a binary protocol, but initially keeping it simple will speed up development.

* **Message Frequency:** We do not want to send frames as fast as the camera produces them (which could be 30 fps) because that would flood the network and the PC. Instead, we will **throttle the preview sending** to a reasonable frame rate. A target of **1–2 fps** for preview should be sufficient to give the operator a sense of the camera view. We might implement this by sending a frame every N milliseconds (e.g. every 500 ms for ~2 fps). This can be done with a scheduled timer or by tracking timestamps in the frame callback (only send if a certain interval has passed since last send). We’ll fine-tune this interval based on performance and network conditions. Optionally, we could make the preview rate configurable or adaptive.
* **Selecting Cameras to Stream:** In a multi-camera setup (visible + thermal), streaming both simultaneously doubles the bandwidth and may not be necessary. We have a couple of options:
* We might **stream only the visible camera’s preview** live, since that’s typically what the operator needs to frame the shot. The thermal camera’s view could be inferred or occasionally checked if needed.
* Alternatively, we could **send both camera frames** but perhaps **alternating**: e.g., send a visible frame, then 500 ms later send a thermal frame, then visible, etc. This way each stream is 1 fps but combined we get 2 fps updates on the PC, one from each sensor.
* Another approach is to **combine the thermal and visible images on the phone** (for example, overlay the thermal image on the visible image or create a side-by-side composite) and send a single blended frame. However, this adds processing on the phone and reduces flexibility on the PC side.

For now, the simplest path is to stream the **visible camera preview** as the main live view. We can later add an option for the thermal view (perhaps toggling which stream to view, or a small PiP overlay). This keeps initial bandwidth usage low and focuses on the primary camera. If needed, we can expand to both streams once the basic system is proven.

## Threading and Performance

Streaming preview frames will be handled on a background thread so that it doesn’t interfere with the app’s UI or the ongoing recording. We need to manage two main tasks in parallel: **capturing frames** and **sending frames**. Here’s how we will organize it:

* **Frame Capture Callbacks:** The ImageReader for the camera preview will provide frames via its OnImageAvailableListener callback, which runs on a background thread (often a handler thread we specify). In that callback, we will acquire the image, and either get the JPEG bytes directly (if the ImageReader is JPEG) or convert it to JPEG (if YUV). We should then **store the latest frame** to be sent. Importantly, we will not try to send from within this callback directly (to avoid blocking the camera callback). Instead, we can use a simple strategy: always keep a reference or buffer for “latest preview frame to send.” When a new frame comes in, if we already have a newer one waiting to be sent, we can discard the older one. We only need the most up-to-date frame.
* **Streaming Thread:** We will have a dedicated **Streaming Thread** (which could be the same as our network client thread if appropriate, or a separate thread) that handles sending data over the socket. This thread can run a loop or timer: every N milliseconds, it checks if a new frame is available to send. If yes, it will take that frame (e.g., copy the byte array) and send it as a JSON message over the network. After sending, it will clear the flag so that we know that frame has been sent. The next camera frame that arrives will set the flag again (or replace the buffer) for the next cycle.
* **Skipping Frames:** By using the above approach, if the camera is producing frames faster than we send, we will **skip/drop frames** and always send the latest one. This is desirable for a preview – it ensures low latency. We prefer to drop older frames rather than queue them, because queued frames would introduce lag (e.g., you could be looking at a several-seconds-old image if network slows down). Our goal is that each transmitted frame is the most recent view. The system will naturally drop frames if the network or PC can’t keep up, similar to how camera pipelines drop frames when overwhelmed[[2]](https://developer.android.com/media/camera/camera2/capture-sessions-requests#:~:text=Parallel%20processing%20suggests%20that%20there,frames%2C%20it%20starts%20dropping%20them).
* **Synchronization:** We will need to handle synchronization between the camera callback thread and the streaming thread. This can be as simple as using a volatile variable or AtomicReference to hold the latest frame bytes, or using synchronized blocks when updating and reading the frame data. We want to avoid complex locking that could stall the camera. Likely we’ll do something like:
* In OnImageAvailable: acquire image -> get JPEG bytes -> replace the global latestFrame byte array (discarding the previous one) -> close the image.
* In the streaming loop: if latestFrame is not null (and/or different from what was sent last time), grab it and send it, then perhaps set a flag that it’s sent (or just send as is, since if another came in it would have overwritten anyway).
* This loose coupling ensures we never block the camera capture (worst case, we drop a frame because the streaming thread is busy).
* **CPU/Memory Impact:** Encoding to JPEG in hardware means minimal CPU. The main CPU load will be from base64 encoding (if we use it) and the networking stack. Base64 encoding a ~50 KB image is not too bad in Java/Kotlin, and doing it at 2 fps should be fine. We should avoid allocating too much; ideally reuse a buffer or use efficient methods. Memory-wise, holding one or two frame buffers (tens of KB each) is trivial for modern phones. We just need to be careful to close ImageReader images promptly to avoid backing up camera buffers.

By structuring it this way, the preview streaming should have minimal impact on the ongoing recording. The recording is likely happening on its own surface (possibly even in hardware via MediaRecorder), and the preview frames we take are low-res and infrequent. We will test on our devices to ensure that adding this stream doesn’t cause frame drops in the main recording. If it does, we might lower the preview resolution or frame rate further.

## Phone UI Indication

While not critical for end users, it’s useful for us (developers) to have some indication on the phone that preview streaming is active and working. This could help during testing to ensure the frames are indeed being captured and sent continuously. Possible UI/UX indications include:

* A small **status icon or text** on the phone’s screen when streaming is on. For example, a “📶 Live” label or an image of a broadcasting icon could appear in a corner of the camera preview UI.
* We might show the **current preview frame rate or size** in a debug overlay. E.g., text like “Streaming 2fps (50KB/frame)” just for our own verification.
* If something goes wrong (e.g., lost connection), this indicator could change or show an alert, but that might be handled in the networking status already.

For initial implementation, a simple blinking dot or a log message might suffice. The main point is to confirm that our loop is running. We will likely add some log outputs (which we can see via logcat) for each frame sent (perhaps throttled logging) – for example: “Preview frame sent: 48 KB”. This will help us debug performance issues. Once stable, a subtle UI indicator can be kept or removed depending on user preference.

By the end of Milestone 2.5, the Android app will continuously stream live video previews to the PC during a recording session. On the PC side, the controller application will receive these frames (via the socket connection), decode them, and display the live video feed for each connected phone. This real-time preview gives the operator immediate feedback on what each phone’s cameras are capturing, allowing verification of framing, focus, and sensor alignment at a glance. It sets the stage for a more user-friendly and safe data collection process, as the operator can catch any issues (like a camera being obscured or misaligned) in real time rather than after the fact.

**Sources:** The approach leverages Android’s Camera2 API for multiple output streams[[1]](https://developer.android.com/media/camera/camera2/capture-sessions-requests#:~:text=A%20single%20Android,more%20than%20one%20stream%20simultaneously) and common techniques for transmitting binary data over sockets (with Base64 encoding overhead of about 33%[[3]](https://en.wikipedia.org/wiki/Base64#:~:text=Base64%20encoding%20causes%20an%20overhead,by%20the%20inserted%20line%20breaks) when embedded in JSON). These choices balance performance and simplicity to achieve live preview streaming with minimal latency and sufficient quality for monitoring purposes.

[[1]](https://developer.android.com/media/camera/camera2/capture-sessions-requests#:~:text=A%20single%20Android,more%20than%20one%20stream%20simultaneously) [[2]](https://developer.android.com/media/camera/camera2/capture-sessions-requests#:~:text=Parallel%20processing%20suggests%20that%20there,frames%2C%20it%20starts%20dropping%20them) Camera capture sessions and requests  |  Android media  |  Android Developers

<https://developer.android.com/media/camera/camera2/capture-sessions-requests>

[[3]](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>