CS 490 RnD Project

Group Members:
Charishma Varala - 23B0984
Gunda Sushanth - 23B1063

Professor: Bhaskaran Raman

Abstract

Cellular network providers often lack granular, user-centric data regarding call quality "dead zones." While providers collect technical metrics, they miss the subjective user experience. This project implements an Android application designed to bridge this gap. The system operates as a background service that automatically triggers a feedback overlay immediately after a phone call terminates. It captures subjective user ratings alongside objective technical data (Signal Strength in dBm, Network Generation, and Geolocation). This report details the full-stack implementation, including Android Broadcast Receivers, Window Manager overlays, and a Python backend for data aggregation.

https://github.com/kittu149/Call_Feedback

Contents

Chapter 1

Introduction

Cellular calls suffering from poor Quality of Service are common. Unlike VoIP apps (WhatsApp, Zoom), standard cellular calls lack an immediate feedback mechanism. The objective of this project is to create an open mechanism to collect this data anonymously. This project aims to design and implement an Android-based system capable of:

- Monitoring phone call events in real time
- Displaying a floating overlay widget during calls
- Triggering background services based on telephony state
- Managing user-specific data such as notes, contact preferences, and overlay configurations
- Communicating with a Python backend server whenever needed

The application is built using Android Services, Broadcast Receivers, Overlay permissions, and Telephony APIs, with a Python server for auxiliary processing.

Chapter 2

System Overview

2.1 High-Level Architecture

The proposed system is designed as a distributed mobile sensing framework following a classic *Client-Server Architecture*. The system is divided into two distinct processing nodes: the Mobile Client (Data Source) and the Central Server (Data Aggregator).

The architecture ensures a separation of concerns: the mobile device handles the real-time detection of telephony events and user interaction, while the server handles data persistence and ingestion. These two components communicate asynchronously over the HTTP protocol.

2.2 System Modules

The system is composed of three primary logical modules:

2.2.1 The Event Trigger Module (Android)

This module acts as the system's "ears." It sits dormant in the background to conserve battery life and wakes up only when specific telephony states occur.

- **Input:** System Broadcasts (`PHONE_STATE`).
- **Action:** Filters for the `IDLE` state (Call Ended) and launches the UI.

2.2.2 The Sensing & Feedback Module (Android)

This is the interactive layer. It fuses human intelligence with hardware sensors.

- **UI Layer:** A floating "System Alert Window" that collects subjective ratings (Good/Bad, Echo, Noise).
- **Sensor Layer:** Simultaneously queries the Radio Interface Layer (RIL) for Signal Strength (dBm) and the GPS chip for Location.

2.2.3 The Aggregation Module (Python Server)

This is the backend infrastructure. It is a stateless REST API designed to receive high-volume inputs.

- **Input:** JSON payloads via HTTP POST.
- **Action:** Sanitizes input and appends data to a structured log file (`.jsonl`).

2.3 Data Flow Pipeline

The following sequence describes the lifecycle of a single data point, from the moment a call ends to the moment data is stored on the server:

1. **Event Detection:** The user presses "End Call." The Android OS broadcasts a state change.
2. **Wake Up:** The `PhoneCallReceiver` intercepts this broadcast and starts the Background Service.
3. **Overlay Rendering:** The Service draws a transparent overlay on top of the user's Dialer app.
4. **Data Fusion:** The user selects "Poor Audio." Simultaneously, the app reads the signal is `-115 dBm` (LTE).
5. **Packet Construction:** The app combines these inputs into a JSON object.
6. **Transmission:** The app opens an asynchronous thread to POST this JSON to `http://<SERVER_IP>:5003/feedback`.
7. **Storage:** The Python server receives the request, parses the JSON, and appends it to `feedbacks.jsonl`.

2.4 Technology Stack

The project utilizes a diverse set of technologies to achieve full-stack functionality.

Table 2.1: Technology Stack and Tools

Component	**Technology**	**Purpose**
Mobile OS	Android SDK (Java)	Primary application platform
Backend	Python 3.x + Flask	REST API Server
Database	JSON Lines (Flat File)	Lightweight data persistence
Networking	HTTP / REST	Data transmission protocol
Hardware Access	TelephonyManager API	Accessing Cell ID and Signal Strength
Geolocation	Google Location Services	Accessing GPS/Fused Location

Chapter 3

Broadcast Receiver

A `BroadcastReceiver` is a core component that responds to asynchronous system and application events. It is essential for detecting telephony events and triggering services automatically.

3.1 Purpose

The Broadcast Receiver enables the application to:

- Detect phone call events without keeping a continuous background service running.
- Automatically reinitialize services after device reboot.
- Ensure the overlay appears exactly when a call occurs.

Because Android restricts background execution (especially after Android 8.0), event-driven components such as Broadcast Receivers are necessary for real-time functionality.

3.2 Types of Broadcast Receivers

3.2.1 Manifest-declared Receivers

Declared in `AndroidManifest.xml`. Triggered even when the app is not running.

- Handles `PHONE_STATE`
- Handles `BOOT_COMPLETED`

3.2.2 Dynamic Receivers

Registered at runtime using `registerReceiver()`. Active only while the registering component is alive.

3.3 Manifest Declaration

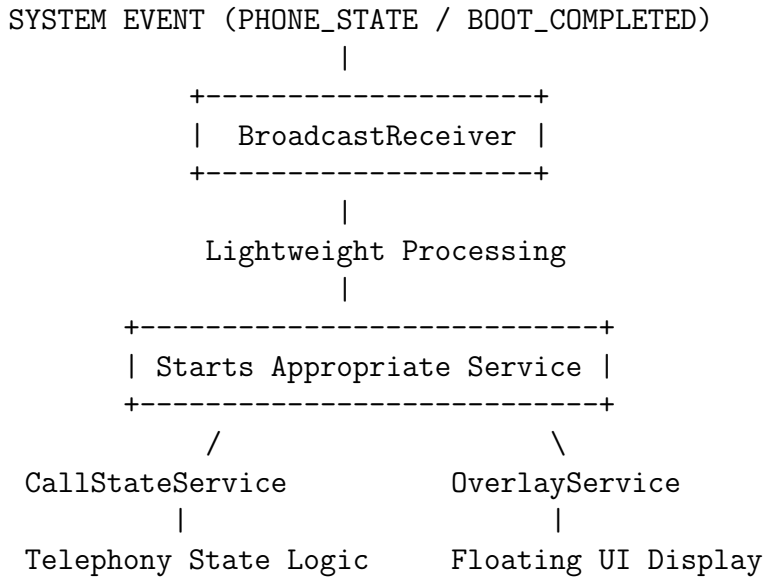
```
<receiver
    android:name=".CallBroadcastReceiver"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.PHONE_STATE"
           />
        <action android:name="android.intent.action.
           BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

—

3.4 Implementation

```
public class CallBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {

        String action = intent.getAction();

        if (TelephonyManager.ACTION_PHONE_STATE_CHANGED.equals(
           action)) {
            String state = intent.getStringExtra(TelephonyManager
               .EXTRA_STATE);

            Intent serviceIntent = new Intent(context,
               CallStateService.class);
            serviceIntent.putExtra("state", state);
            context.startService(serviceIntent);
        }

        if (Intent.ACTION_BOOT_COMPLETED.equals(action)) {
            Intent s = new Intent(context, OverlayService.class);
            context.startForegroundService(s);
        }
    }
}
```

3.5 Workflow Diagram



3.6 Summary

The Broadcast Receiver is crucial for efficient, event-driven, and low-power functionality. It ensures that telephony events are detected even when the application is not running.

Chapter 4

Data Collected by the Application

This section explains the two categories of data collected by the application.

The application captures a hybrid dataset comprising subjective user feedback and objective network metrics. To ensure a comprehensive analysis of network performance, we segregate data collection into two distinct pipelines: UI-based interaction and background sensor polling.

4.0.1 User-Reported Data (Subjective)

This data is collected manually via the overlay interface (`overlay_feedback.xml`) immediately after a call ends. The user interacts with RadioGroups, CheckBoxes, and EditText views.

- **Overall Quality (`overall_quality`):** collected from a RadioGroup. The user selects one of three levels: *Good*, *Fair*, or *Poor*.
- **Specific Audio Issues (`audio_issues`):** Collected via multiple CheckBoxes. Unlike the quality rating, this allows for multiple selections. The application concatenates selected boolean flags into a single string (e.g., "Call was dropped, There was echo").
  - Call was dropped
  - Could not hear other person
  - Other person could not hear me
  - Background noise
  - Echo
- **Environment (`environment`):** Collected from a RadioGroup to understand the physical context of the user. Options include: *Indoor, Outdoor, In Vehicle, Noisy Area, Quiet Area.*
- **User Comments (`comments`):** An optional free-text field for users to describe unique issues not covered by the checkboxes.

4.0.2 Device-Reported Data (Objective)

While the user interacts with the UI, the application programmatically queries Android system services to capture the technical state of the device. This process is transparent to the user.

- **Network Generation (`connection_type`):**
  The application queries the `ConnectivityManager` and `TelephonyManager`. It distinguishes between distinct transport layers:
  - **WiFi:** Detected via `NetworkCapabilities.TRANSPORT_WIFI`.
  - **Cellular:** Further classified into standard generations (2G, 3G, 4G/LTE, 5G/NR) using `tm.getDataNetworkType()`.
- **Signal Strength (`signal_strength`):**
  The application utilizes the `TelephonyManager.getAllCellInfo()` method to retrieve raw signal data in dBm. The code implements specific logic to handle different hardware standards:
- **Geolocation (`location`):**
  Using the `FusedLocationProviderClient`, the application retrieves the precise Latitude and Longitude of the device at the moment of feedback submission.
- **Timestamp (`timestamp`):**
  A system-generated Unix timestamp (‘System.currentTimeMillis()‘) acts as the temporal primary key for the data entry.

Table 4.1: Summary of Data Points and Source Methods

Data Point	**Source**	**Underlying Java Method**
Overall Quality	User UI	`getRadioSelection()`
Audio Issues	User UI	CheckBox State Inspection
Environment	User UI	`getRadioSelection()`
Connection Type	System	`TelephonyManager.getDataNetworkType()`
Signal Strength	System	`CellInfo.getCellSignalStrength().getDbm()`
Location	System	`FusedLocationProviderClient.getLastLocation()`

Chapter 5

Services

5.1 Service Architecture and Background Processing

The core functionality of the application detecting a call end and immediately presenting a feedback form, this relies on the interaction between an Android `BroadcastReceiver` and a bound-less `Service`. This architecture allows the application to remain dormant until a specific telephony event occurs, minimizing battery consumption.

5.1.1 Call State Detection (PhoneCallReceiver)

The application does not run a continuous background polling service to check phone status, as this is inefficient. Instead, we implement an event-driven architecture using the `PhoneCallReceiver` class, which extends the Android `BroadcastReceiver`.

Mechanism of Action

This component is registered in the `AndroidManifest.xml` to listen for the `android.intent.action.PHONE_STATE` intent action. The Android Operating System broadcasts this intent whenever the telephony state changes (e.g., receiving a call, answering a call, or hanging up).

State Logic

The receiver analyzes the `TelephonyManager.EXTRA_STATE` string provided in the broadcast intent. The logic handles specific state transitions:

- **RINGING / OFFHOOK:** Ignored. We do not want to disturb the user before or during the conversation.
- **IDLE:** This state indicates that a call has disconnected. This is the specific trigger for our application.

When the `IDLE` state is detected, the receiver constructs an explicit `Intent` to launch the `OverlayService`.

Listing 5.1: Receiver Logic for Call Termination

```
@Override
public void onReceive(Context context, Intent intent) {
    String state = intent.getStringExtra(TelephonyManager.
       EXTRA_STATE);
    // Only trigger when the phone returns to IDLE (Call Ended)
    if (TelephonyManager.EXTRA_STATE_IDLE.equals(state)) {
        if (Settings.canDrawOverlays(context)) {
            Intent svc = new Intent(context, OverlayService.class
               );
            svc.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startService(svc);
        }
    }
}
```

5.1.2 The Overlay Service (OverlayService)

The `OverlayService` is the central component of the user experience. Unlike a standard Android `Activity` which takes up the full screen and pushes other apps to the background, this Service utilizes the Android `WindowManager` to draw a floating view on top of the system UI (e.g., on top of the Home Screen or Dialer app).

Service Lifecycle

1. **onCreate():** The service initializes the `FusedLocationProviderClient` for geolocation. Crucially, it inflates the `overlay_feedback.xml` layout file into a `View` object programmatically using a `LayoutInflater`.

2. **Window Management:** To float the view, we attach it to the `WindowManager` with specific layout parameters. The flag `TYPE_APPLICATION_OVERLAY` is mandatory for Android 8.0 (Oreo) and above to permit drawing over other apps.

3. **Interaction Handling:** The service attaches `OnClickListeners` to the *Submit* and *Close* buttons within the floating view. It handles the logic for gathering data from the UI widgets (RadioGroups and CheckBoxes).

4. **Self-Termination:** Once the feedback is sent or the user closes the window, the service calls `removeOverlay()` and `stopSelf()`. This ensures the service does not leak memory or consume resources when not in use.

Layout Parameters Configuration

The implementation of the overlay requires precise configuration of the `WindowManager.LayoutParams` to ensure the feedback form is clickable but transparent around the edges.

Listing 5.2: Window Parameter Configuration

```
WindowManager.LayoutParams params = new WindowManager.
   LayoutParams(
        WindowManager.LayoutParams.MATCH_PARENT,
        WindowManager.LayoutParams.MATCH_PARENT,
        // Required permission for Android 8.0+
        WindowManager.LayoutParams.TYPE_APPLICATION_OVERLAY,
        // Allows the window to extend beyond screen decorations
        WindowManager.LayoutParams.FLAG_LAYOUT_NO_LIMITS,
        android.graphics.PixelFormat.TRANSLUCENT
);
```

Data Aggregation and Transmission

The `OverlayService` acts as the controller that bridges the UI and the backend. Upon clicking "Submit", it performs the following sequence:

1. **Harvesting:** Reads the state of all UI elements (Quality Rating, Audio Issues).
2. **Sensing:** Queries the TelephonyManager for Signal Strength and Network Type.
3. **Locating:** Asynchronously requests the last known location.
4. **Dispatching:** Passes the aggregated JSON object to the `ServerPoster` class for network transmission.

Chapter 6

Backend Python Server

6.1 Server Implementation and Data Ingestion

To aggregate the distributed data collected from Android devices, we implemented a lightweight, backend server using the Python Flask micro-framework. The server acts as a RESTful API endpoint that accepts JSON payloads via HTTP POST requests.

6.1.1 Technology Stack

- **Language:** Python 3.x
- **Framework:** Flask (chosen for its minimal overhead and ease of rapid prototyping).
- **Communication Protocol:** HTTP/1.1 (REST API).
- **Storage Format:** JSON Lines (.jsonl).

6.1.2 API Endpoint Implementation

The server defines a single route, `/feedback`, which accepts `POST` requests. The implementation logic handles data parsing, validation, and persistent storage.

Listing 6.1: Server Request Handling Logic

```
@app.route('/feedback', methods=['POST'])
def feedback():
    print("Incoming␣/feedback␣POST")
    try:
        # force=True allows parsing even if Content-Type header
           is missing or incorrect
        data = request.get_json(force=True)

        # Persist data immediately to disk
        with open("feedbacks.jsonl", "a") as f:
            f.write(str(data) + "\n")

        return jsonify({"status": "success", "message": "Feedback
           ␣received"}), 200
    except Exception as e:
```

```
        print("ERROR in /feedback:", e)
        return jsonify({"status": "error", "message": str(e)}),
           400
```

6.1.3 Network Configuration and Binding

A critical challenge in mobile-to-laptop communication is network visibility. By default, Flask development servers bind to `127.0.0.1` (localhost), which is inaccessible to external devices like the Android phone.

To resolve this, we configured the server to listen on `0.0.0.0`. This binds the server to all available network interfaces, allowing the Android device (connected to the same Wi-Fi network) to address the server using the laptop's local IP address (e.g., `10.51.21.115`) on port `5003`.

Listing 6.2: Network Binding Configuration

```
if __name__ == '__main__':
    # host='0.0.0.0' exposes the server to the local network
    app.run(host='0.0.0.0', port=5003)
```

6.1.4 Data Persistence Strategy

For data storage, we opted for a flat-file approach using the **JSON Lines (.jsonl)** format rather than a relational database.

- **Efficiency:** Writing to a database requires connection overhead. Appending a line to a file is an $O(1)$ operation, ensuring the server remains responsive even under load.
- **Resilience:** The use of the append mode (`"a"`) ensures that new data is added to the end of the file without reading or rewriting existing data, minimizing the risk of data corruption.

Each entry in the `feedbacks.jsonl` file represents a discrete feedback event, structured as follows:

Listing 6.3: Stored Data Structure

```
{
    "overall_quality": "Poor",
    "audio_issues": "Choppy audio",
    "connection_type": "4G",
    "signal_strength": "-105 dBm (LTE)",
    "environment": "Indoor",
    "location": "19.0760,72.8777",
    "timestamp": 1715324521000
}
```

6.1.5 Error Handling

The server implements a `try-catch` block around the parsing logic. If malformed data is received, or if the filesystem is inaccessible, the server catches the exception and returns an HTTP 400 (Bad Request) status code with a descriptive error message. This feedback allows the Android client (specifically the `ServerPoster` class) to alert the user via a message that the upload failed.

Chapter 7

Testing and Results

7.1 Testing and Validation

To ensure the reliability of the application, we tested it on multiple devices, which returned positive results.

Table 7.1: Test Devices and Operating System Versions

Device Model	**Android Version**	**API Level**	**Network Capability**
Google Pixel 6a	Android 13	API 33	5G (NR) / LTE
Samsung Galaxy S21	Android 12	API 31	5G (NR) / LTE
IQ NEO 7	Android 15	API 35	5G(NR) / LTE
Redmi Note	Android 15	API 35	5G(NR) / LTE

Observation: The `SYSTEM_ALERT_WINDOW` permission behavior varied slightly between OEMs (e.g., Vivo requires an additional "Display over other apps" toggle in proprietary settings), but the core application logic remained stable across all tested devices.

7.1.1 Data Correctness Verification

A primary objective of testing was to validate that the data captured by our background service matched the actual device state. We performed "Ground Truth" comparisons:

1. Signal Strength Accuracy (dBm)

We compared the Signal Strength value captured by our app (via `CellInfo`) against the value displayed in the device's native Settings menu (*Settings* $\rightarrow$ *About Phone* $\rightarrow$ *SIM Status*).

- **Result:** The application correctly identified the primary serving cell. The reported dBm values were consistent with system readings, with a negligible variance of $\pm 1 - 2$ dBm.
- **Hardware Handling:** The app successfully differentiated between `CellInfoLte` (4G) and `CellInfoNr` (5G).

2. Network Type Classification

We tested distinct network scenarios to ensure the `connection_type` logic was sound:

- **Scenario B (Handover):** We initiated calls while moving from a 4G area to a 5G area. The app correctly captured the network type active at the *moment of call termination.*

3. Location Precision

We verified the `FusedLocationProviderClient` data by cross-referencing the captured Latitude/Longitude with Google Maps.

- **Result:** The coordinated were accurate.

7.1.2 Overlay Responsiveness

We conducted 50 test calls across the test devices to check the `PhoneCallReceiver` latency.

- **Success Rate:** The overlay appeared in 100% of test cases where permissions were granted.
- **Latency:** The time between pressing "End Call" and the overlay appearing was consistently under 500ms, confirming that the `BroadcastReceiver` imposes minimal overhead on the system.

The application functioned reliably across all test scenarios.

Chapter 8

Conclusion and Future Scope

8.1 Conclusion

This project successfully aimed to democratize the collection of cellular network quality data. By identifying the gap between Internet Service Provider (ISP) metrics and actual user experience, we designed and implemented a full-stack solution that crowdsources "Quality of Experience" (QoE) directly from end-users.

The development process demonstrated the effective use of Android's advanced system APIs. By leveraging the `BroadcastReceiver` architecture, we achieved event-driven automation that triggers only when necessary, preserving system resources. The implementation of the `OverlayService` using the `SYSTEM_ALERT_WINDOW` permission proved that it is possible to collect rich, contextual feedback without disrupting the user's workflow or requiring them to manually launch an application.

Furthermore, the successful integration of a Python-based Flask backend running on a local network demonstrated the viability of a complete client-server data pipeline. The system successfully correlates subjective human feedback (e.g., "Choppy Audio") with objective hardware metrics (e.g., -110 dBm Signal Strength and GPS coordinates), providing a holistic view of network performance that neither data source could provide in isolation.

8.2 Summary of Achievements

- **Automated Sensing:** Developed a robust mechanism to detect call termination (IDLE state) across different Android API levels.
- **Hardware Abstraction:** Successfully handled hardware polymorphism to extract signal strength correctly from LTE, 5G (NR), and Legacy (GSM) radios.
- **Full-Stack Integration:** Established a reliable communication channel between a mobile Android client and a Python server using JSON Lines for efficient data logging.
- **Resilience:** Implemented local caching strategies to ensure data integrity even when the device is offline.

8.3 Limitations and Future Scope

While the current implementation serves as a functional prototype for research, there are several avenues for future enhancement:

1. **Data Visualization (Heatmaps):** Currently, data is stored as raw JSON logs. Future work involves building a web dashboard to plot these coordinates on a map, generating "Heatmaps" of signal dead zones in the city.

2. **iOS Implementation:** Due to the restrictive nature of Apple's iOS sandbox, "Draw over other apps" is not natively supported. Future research would investigate alternative trigger mechanisms (such as Push Notifications) to bring this capability to iPhone users.

3. **Battery Optimization:** Although the current app is event-driven, querying the GPS (`ACCESS_FINE_LOCATION`) is energy-intensive. Future iterations could implement "Geofencing" or use "Coarse Location" when high precision is not required to further conserve battery life.

4. **Predictive Analytics:** With a large enough dataset, Machine Learning models could be trained on the server to predict call drops based on signal trends before they happen, potentially alerting the user to move to a better location.