EECE 451: Mobile Networks & Applications

Dr. Ali Hussein

Network Cell Analyzer Project Report

Samia Noaman, Omar Ramadan, Lama Hasbini, Reem Arnaout

April 2024

Table Of Contents

Contents

Abstract

The increasing demand for improved network performance in mobile devices necessitates the development of efficient tools for analyzing cellular network data. In response to this need, we propose the development of an Android application titled "Network Cell Analyzer" to facilitate the analysis of cell-specific data received from the serving base station of the connected cellular network. This application aims to track the history of network operators and cellular network types to which mobile devices are connected, providing valuable insights into link quality and measurements in a distributed manner.

The proposed system leverages smartphone APIs to collect cell information from actively connected base stations, supporting GSM/GPRS/EDGE (2G/2.5G), UMTS (3G), and LTE (4G) networks. Through a set of methods provided by the Android API, the application acquires cell-related information, including operator details, signal power, network type, frequency band (if available), cell ID, and timestamps. This data is regularly transmitted to a designated server, which is designed to receive network cell data from multiple running phones at regular intervals.

The server component of the system is responsible for storing the received data in a database and generating statistics based on requests from the Android application. These statistics include average connectivity time per operator and network type, average signal power per network type and device, and average SNR or SINR per network type (when applicable). Users have the flexibility to specify time periods for which statistics are calculated, enabling customized analysis based on their requirements. Additionally, the server interface provides centralized statistics such as the number of connected mobile devices and the IP and MAC addresses of previously and currently connected devices. This interface enhances the monitoring and management of network data, offering valuable insights into the performance and connectivity of mobile devices within the network.

1 Introduction

The "Network Cell Analyzer" is an Android mobile app designed using Android studio to extract and analyze data received from base station to which the android device is connected. Our app supports 2G/2.5G , 3G and 4G technologies. The project also includes a server that will receive cell data from the running devices each 10 seconds. This server will be receiving regularly the cell information already extracted by the application. While the server should be accepting multiple connections from different mobile applications at the same time, it will save the data received in a database providing statistics related to this data upon request. In addition, the Android application can request statistics from the server, which analyzes the data entries populating its database and evaluate the required statistical values before sending them back to the application's user interface. Finally, the server displays more information on its user interface, such as the number of connected devices as well as the IP and the phone ID of current and previously connected devices.

2 Literature Review

The cellular network is a crucial yet "closed" system of communication. On the one hand, mobile consumers are using their 2G/2.5G/3G/4G networks on their smart devices—such as tablets and smartphones—to access internet services more frequently. With an exponential rise in traffic, the resulting data volume is expected to reach 97% by 2019 and account for 88% of all mobile traffic globally. However, on all cellular protocols, users' and devices' access to their runtime activities is severely restricted. Using the socket API, mobile apps send data across the cellular interface. Beyond that, users can still essentially remain in the dark about the network itself. How cellular protocols function at the device and inside the network is difficult for academics and developers to precisely comprehend and improve due to closed access to delicate runtime network activities. For instance, when a handoff occurs while the device is in use, it is unaware of the reason behind it or if it was the right choice. In actuality, even when 4G is available, the device has been known to switch over and become stuck in 2G. Figure 1 illustrates a simplified cellular network architecture.The mobile device (i.e., smartphone) connects to the Internet or telephony network through the base stations and the cellular core network. Both control-plane and data-plane (i.e., user plane) operations are needed to receive data/voice services. The data plane delivers user content (data/voice), whereas the control plane exchanges signaling messages to facilitate content delivery. Cellular network protocol stack. Figure 1 (middle) shows the cellular protocol stack at the device, which has three parts. The first is to enable radio access between the device and the base station. Physical (L1) and link (L2) functionalities, including PHY, MAC, RLC (Radio Link Control) and PDCP (Packet Data Convergence Protocol), are implemented. The second part is the control-plane protocols, which are split into access stratum (AS) and non-access stratum (NAS). The

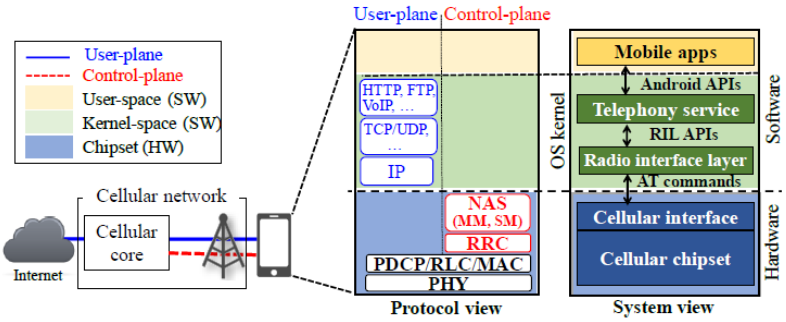

Figure 1: Network architecture (left), protocol stack (middle), and information access at device (right).

AS protocols regulate radio access through Radio Resource Control (RRC). RRC is mainly for radio resource allocation and radio connection management; it also helps to transfer signaling messages over the air. NAS is responsible for conveying non-radio signaling messages between the device and the core network. Two protocols of mobility management (MM) and session management (SM) also belong to the control plane. MM offers location updates and mobility support for call/data sessions, while SM is to create and mandate voice calls and data sessions. The last piece is the data-plane protocols above IP, which are not cellular specific but use the standard TCP/IP suite. [1]

3 Methodology And Architecture

3.1 Project Architecture

Figure 2 is a visual overview of the Network Cell Analyzer application architecture. The mobile application is developed using Android Studio with Java, focusing on extracting comprehensive network cell information. This data includes the network operator, signal power, signal-to-noise ratio (SNR), network type (such as GSM, UMTS, and LTE), cell ID, phone ID, frequency, and timestamps indicating when the connection occurred.

The server-side component is developed using Flask, a lightweight web framework for Python. Communication between the mobile app and the server is established through a TCP socket connection, allowing the client (mobile app) to send network-related information to the server for processing. Upon receiving this data, the server inserts it into a database managed by SQLAlchemy, a popular object-relational mapping (ORM) tool for Python. Once the data is stored in the SQLAlchemy database, various queries are executed to extract specific statistics. These statistics can include averages, counts, and other metrics that provide insights into network performance and user activity. The server then

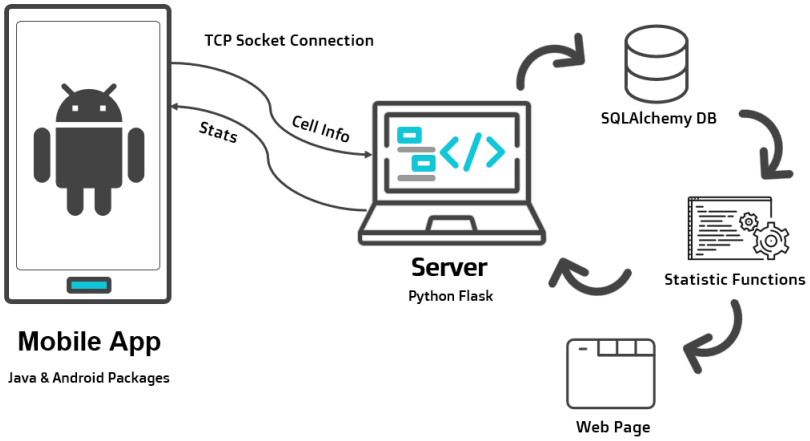

Figure 2: Network Cell Analyzer project architecture.

returns these statistics to the mobile application, where they are used to populate the dashboard and other sections of the user interface.
Additionally, these statistics are displayed on a webpage, providing a broader view of the data for monitoring and analysis. The combination of mobile data extraction, server-side data storage and processing, and real-time updates to the user interface creates a robust and scalable architecture for analyzing network cell information and presenting it in a user-friendly manner. This structure supports future scalability, allowing for the integration of additional features and deeper data analysis as the project evolves.

3.2 2G/2.5G/3G/4G Cell Information Querying

Starting with gathering the required cellular information from corresponding Java-based Android packages, we began by creating our XML files where we defined TextView components in order to display the cell information we will extract, hence visualizing the queried cellular information. This was first done on the `activity_main` XML layout, but was later moved to the `fragment_home` XML layout, i.e. the first page of the application's user interface allocated for live cell information querying. Note that the TextView components are identified by their unique ID values, and thus will be accessed for modification using functions like `findViewByID` (for static TextView modification) or `setText` (for dynamic data binding).
Next, we have implemented methods in the `MainActivity` class that allow for checking and requesting the permissions needed for correct operation of the application. Hence, the `checkPermission` method uses `ActivityCompat.checkSelfPermission()` to determine if the permissions `ACCESS_FINE_LOCATION` and `READ_PHONE_STATE` are granted or not. These permissions are critical for our mobile application for the following reasons:

- **ACCESS_FINE_LOCATION**: This permission allows the application to access the precise location of the device, which is crucial for tower identication and signal mapping. Thus, it helps determining which cell tower the device is connecting to and analyzing the cellular network in that specific area.
- **READ_PHONE_STATE**: This permission allows the application to access telephony-related information such as the network operator, cell ID, network type, signal strength, frequency, etc. It also allows for device identification, as it allows for requesting the unique identifier of an operating mobile device.

Hence, these permissions are fundamental for the application we are developing, as they grant access to the information needed to analyze cellular networks and understand how they interact with mobile devices.

Next, we have defined a `UIDataExtractor` class, where all the needed cellular information will be queried from the mobile device using `TelephonyManager` and `CellInfo`-based android packages and APIs. Hence, this class is based on several static methods, each responsible for obtaining one information from the cellular network. Note that each method takes a `Context` parameter, which is used to access system services, interact with other user interface-related components, or obtain system resources. Each method is defined for one specific purpose, as explained below:

- **getOperator()**: This method returns the network operator name, i.e. the carrier's name (e.g.: Alfa, touch, T-Mobile, Verizon, AT&T, etc.) Note that using Android Studio, the network operator detected by the built-in emulator might be limited to "Android".
- **getCellID()**: This method returns the Cell ID, which identifies the specific cell tower the operating device is connected to. It fetches all cell information using `manager.getAllCellInfo()`, which provides a snapshot of all cell towers and related information accessible by the device, and then iterates over the resulting list `cellInfoList` to find the appropriate cell ID based on the network type (GSM, WDCMA, LTE).
- **getNetworkType()**: This method returns the network type (e.g., GPRS, EDGE, GSM, UTMS, HSDPA, LTE, NR). It uses a helper method of the same name (`getNetworkType(int)`) that translates the numeric network type into a human-readable string.
- **getSignalStrength()**: This method returns the signal strength in dBm (decibel milliwatts). It also retrieves all cell information, and extract from `cellInfoList` the signal strength based on the cell type (GSM, WDCMA, LTE).

- **getSNR()**: This method returns the Signal-to-Noise Ratio (SNR) for LTE-based networks. If the cell type is identified as GSM or WDCMA, it returns "NONE" since SNR does not apply for these technologies.

- **getDate()**: This method returns the current date and time. It uses `SimpleDateFormat` to format the current date into the following human-readable form: `Year-Month-Day Hour-Minute-Second`. This acts as the timestamp when recording the time of the retrieved cellular information and storing them in the server's database.

- **getFrequency()**: This method returns the frequency band information for WCDMA and LTE networks. It calculates the frequency band based on the cell type.

- **getPhoneID()**: This method returns the Android device ID, which is a unique identifier retrieved from `Settings.Secure`. This identifier remain consistent accross app sessions unless the device is reset to factory settings, and thus compensates for the MAC address that cannot be programmatically accessed by non-system applications for data protection and safety reasons. [2]

Note that most of the above-mentioned methods contain a `TelephonyManager` object `manager`. This is obtained by invoking `context.getSystemService()`, where a system service provided by Android is retrieved per static invoking of a `context` (Activity, Service, Fragment or Application). The `TelephonyManager-.class` parameter allows to specify which system service to retrieve from the `getSystemService()` method, and then casts the retrieved generic `Object` type into `TelephonyManager`. After `manager` is obtained in its correct `TelephonyManager` type, the needed cellular information can thus be queried using Android libraries and packages.

3.3 JSON Operations for Data Handling

Before delving into the technicalities of implementing the client and server-related codes, we will briefly explain the format employed in storing and retrieving the queried cellular information in the server's database. We have used JSON format, which is a lightweight format that is human-readable and compatible for machines-parsing. It is often used for transmitting data between a server and a web/mobile application, and thus can be easily applied in this Network Cell Analyzer project. It represents structured data using key-value pairs enclosed in curly braces `{}`. It follows the format `{key:value}`, where `key` is of type `string` and `value` is of type `string, array, object, integer, float,` etc. or `null`. This structure allows complex data representation, including nested objects and arrays.

In the MainActivity class, we have implemented several methods to format, validate and represent data using JSON:

- **formatData()**: This method creates a JSON object by gathering information from various `HomeViewModel` and `DashboardViewModel` objects, and then populates the JSON object with key data (phone ID, signal power, SNR, frequency band, time, cell ID, network type, operator, date 1 & date 2). The method returns a string representation of the JSON object. Note that this JSON format supports both sending the queried cellular information and retrieving statistics from the server, which will be elaborated further in the upcoming sections.

- **parseAndValidateJson()**: This method takes a JSON string and an array of mandatory keys. It parses the string into a JSON object `jsonObject`, and only returns it after ensuring that all mandatory keys are present.

- **displayData()**: This method updates the dashboard fragment with information extracted from a JSON object (i.e. the average connectivity time per operator, average connectivity time per network, average signal power per network type, average SNR per network type, and average signal power per device), after formatting them using `prettyJson()`. The formatted data is then used to update the user interface via the `dashboardViewModel`. Note that the JSON object provided to this method is received from the server, and thus is not populated by the cellular information retrieved by the `HomeFragment`, which explains the discrepancy in the JSON keys and values between this method's JSON object and that of `formatData()`.

- **prettyJson()**: This method takes a JSON string and formats it into a more readable structure. It parses the JSON string into a `JSONObject`, iterates through its keys, and appends each key-value pair to a `StringBuilder` with appropriate formatting, before returning the formatted string.

Collectively, these methods handle the formatting, validation, and presentation of JSON data within our Android application. They ensure the proper processing of data, allowing seamless extraction, validation, and display of key information, ultimately contributing to a robust user experience.

3.4 Server Design and Functionality

In order to store the queried cellular information in a database and retrieve the corresponding statistical results, we have implemented a Python-based server (`app.py`) that utilizes Flask, a lightweight web framework, along with other libraries to manage cellular network data and facilitate communication with client applications.
The script imports several modules, including Flask for web serving, Flask SQLAlchemy for database operations, and other utility modules like `datetime`, `socket`, `threading` and `json`. In addition, upon creating the Flask application, the database configuration is set to use SQLite, with the database (`test.db`) located in the same directory as the server. SQLAlchemy is initalized to manage

database transactions.

The database model is defined using the `NetworkData` class, which represents a table in the SQLite database. This table contains columns to store several attributes related to cellular network information querying, such as `date_created`, `phoneID`, `ip_address`, `operator`, `signal_power`, `snr`, `network_type`, `frequency_band`, and `cell_id`. Note that the `id` column is the primary key, the `date_created` is set to the current date and time (i.e.: date and time of querying and storage of network information), and other columns represent different types of data (text, integer, etc.). Note that while the `ip_address` is not displayed on the mobile application's user interface, it is a pivotal information that should be displayed on the server's user interface, and thus should be stored within each database entry.

Two utility functions are defined, notably `parse_and_validate_json()` and `safe_string_to_int()`, which play key roles in handling and validating JSON data and converting strings to integers with error handling. More specifically:

- **parse_and_validate_json()**: This function parses a given JSON string into a JSON object and checks whether certain mandatory fields are present. It is designed to validate incoming JSON data and ensure it contains all required information before further processing. The function uses `json.loads()` to parse the JSON string into a Python dictionary (`'data'`), and then checks whether all mandatory fields are included in `'data'`. If they are, the function returns the parsed JSON object, which helps in maintaining data integrity and avoiding downstream errors.
- **safe_string_to_int()**: This function converts a string to an integer, with a default value if the conversion fails. It is designed to prevent exceptions caused by invalid string-to-integer conversions, and thus handles problematic data without causing the server to crash, hence enhancing its robustness and resilience.

In addition, several routes and functions have been implemented in our Flask-based server to handle client requests, manage socket connections, and interact with a SQLAlchemy database. These routes provide the web-based interface for users, while the functions process incoming data, update the database, and calculate various statistics for cellular network analysis. The key components include endpoints for rendering web pages, handling client connections, and performing background tasks, each serving a specific role in the overall architecture, as explained in the following:

- **index()**: This function is a route for the main page of the Flask server. It renders an HTML template `'index.html'` with statistics and the count of active connections.
- **handle_client()**: This function handles incoming client connections through sockets. It manages active connections, receives data, validates it, and interacts with the database. This function will be revisited in Section 3.4.

- **calculate_statistics() and calculate_statistics_all()**: These two functions calculate statistics based on the data stored in the `NetworkData` table. These functions perform various operations to compute averages and summarize the data. Notable statics calculated by these functions are the average connectivity time per operator, average connectivity time per network type, average signal power per network type, average signal power per device, and average SNR per network type. These functions also handle invalid data by invoking `safe_string_to_int()` and thus avoiding exceptions. Note that the functions differ by the scope and data they operate on. Thus, `calculate_statistics()` takes a start date and end date as parameters in order to compute the above-mentioned statistics based on data within that range, while `calculate_statistics_all()` does not take any parameter, as it calculates the same statistics for the entire dataset. In other words, the former is used in the application's `DashboardFragment`, where the user enters the two specific dates between which they deem to retrieve statistics about the server, while the latter is used in the server's HTML template to display the statistics of all the database entries.

- **socket_server()**: This function listens for incoming client connections on a specific port (8080), starts a new thread for each connection, and uses `handle_client()` to process client data. This function will be revisited in Section 3.4.

- **run_statistics_update()**: This function periodically updates statistics by calculating them at regular intervals. Specifically, it sleeps for 5 seconds after invoking the (`calculate_statistics_all()`) function, all while running in a continuous while loop. Hence, this function enables real-time data processing and ensures the server has the latest statistics available.

These elements work together to ensure a seamless user experience while maintaining the integrity and efficiency of server-side operations.

Finally, when `app.py` is run, it starts the `socket.server` and `run_statistics_update` in separate threads to handle client connections and maintain real-time statistics updates. The Flask application is then run on port 8000 of the localhost, which can be accessed via any web browser with basic HTML capabilities, hence visualizing the simple user interface created for the server.

3.5 Mobile / Server Communication

TCP (Transmission Control Protocol) is a widely used communication protocol in computer networks. It operates at the transport layer of the Open Systems Interconnection (OSI) model and provides reliable, connection-oriented communication between two hosts over an IP network. Below is a brief rundown of a typical TCP connection:

- The Three-Way Handshake:
    1. **SYN**: The client sends a SYN (synchronize) packet to the server to initiate the connection.
    2. **SYN-ACK**: The server responds with a SYN-ACK (synchronize-acknowledgment) packet, indicating it received the client's request and is willing to establish a connection.
    3. **ACK**: The client acknowledges the server's response by sending an ACK packet. At this point, the TCP connection is established, and both parties can start sending data.
- Data Transfer: After the connection is established, data can be exchanged bidirectionally between the client and server. TCP ensures reliable delivery by implementing features such as sequence numbers, acknowledgments, and retransmissions.
- Connection Termination:
    1. **FIN**: Either the client or server sends a FIN (finish) packet to initiate the connection termination.
    2. **ACK**: The receiving party acknowledges the FIN packet.
    3. **FIN**: The receiving party sends its own FIN packet to indicate its agreement to terminate the connection.
    4. **ACK**: The initiating party acknowledges the FIN packet, and the connection is closed.

When it comes to the TCP connection between our server (`app.py`) and the clients (`MainActivity.java`), three functions deal with this:

- **socket_server()**: This function is found in `app.py`, and it sets up a TCP server to listen for incoming client connections. It creates a new TCP socket using `socket.AF_INET` (IPv4) and `socket.SOCK_STREAM` (TCP), then binds it to IP address `0.0.0.0` and port `8080`. The server uses `listen()` to accept up to five pending connections. When a client connects, the `accept()` method blocks until the connection is established, then returns a new socket representing the client connection (`client_socket`) and the client's address (`addr`). At this point, a new thread (`client_thread`) is created to handle the client connection concurrently. The thread uses `handle_client()` as its target function, passing the `client_socket` to process the connection. This design allows the server to manage multiple client connections at once. Successful connections are logged to track server activity and ensure proper operation.
- **handle_client(client_socket)**: This function is designed to manage individual client connections in a multi-client environment. It retrieves the client's IP address using the `getpeername()` method, prints it to the console, and checks if it's already in the `active_client_sockets` list. If it's

a new client, the function adds the IP address to the list and updates the count of active connections. The function then enters a `while True` loop to continuously receive and decode data from the client. If no data is received, it sends a newline character back to the client to keep the connection alive but breaks out of the loop. When the loop ends, the function logs that the connection is closing and releases any resources associated with the connection, ensuring proper cleanup.

- **sendDataRunnable**: Located in `MainActivity.java`, it is a Runnable task responsible for sending data to a server. It specifies the server's IP address and port, matching the settings used in the `socket_server()`. The task uses `formatData()` to prepare the data for transmission. It creates a new TCP socket to connect to the server using the given IP address and port. A timeout of 500 milliseconds is set to prevent the socket from hanging during data transmission. Once the socket connection is established, the prepared data is converted to bytes and sent to the server via the socket's output stream. The `flush()` method is used to ensure all data is immediately sent, without buffering. After sending, the output stream and socket are closed to release resources. If any `IOException` occurs during socket operations, the `catch` block logs an error message indicating the problem.

3.6 Real-time and Statistical Services Provided Using the Android App

Our application extracts various real-time statistics using `formatData()` method which creates a JSON object containing various data points required for the statistics. The `displayData(JSONObject jsonOject)` method will receive a JSON object containing the statistics data. From the JSON object, it extracts the following relevant statistics and updates the UI:

- Average Connectivity Time per Operator
- Average Connectivity Time per Network Type
- Average Signal Power per Network Type
- Average Signal Power per Device
- Average SNR per Network Type

Each specified average is acquired by collecting data over the specified time period, calculating the total sum, then dividing the sum by the total duration of the time period.
These averages are the core reason why the network cell analyzer application is beneficial, particularly in the context of mobile network performance analysis and optimization. The following explains why one might want to know such averages:

- **Network Performance Monitoring**: By tracking averages such as connectivity time, signal power, and SNR, network operators can monitor and evaluate overall network performance. This helps them identify areas for improvement and ensure users receive satisfactory service.
- **Quality of Service (QoS) Assessment**: Operators can use averages to assess the quality of service for different networks or locations. This data guides adjustments to maintain or improve service quality, ensuring consistent user satisfaction.
- **Network Optimization**: By analyzing average signal power and SNR, operators can optimize networks for better coverage and performance. This involves identifying areas with weak signals or high interference and implementing corrective measures like new towers or antenna adjustments.
- **User Experience Improvement**: Insights from connectivity times and signal strengths allow operators to enhance the user experience. They can resolve connectivity issues, optimize network handoffs, and ensure consistent service across various regions.
- **Resource Allocation**: Knowing network performance averages helps operators allocate resources efficiently. They can focus investments on high-demand or underperforming areas, improving cost-effectiveness and overall network quality.
- **Competitive Analysis**: Comparing performance averages with competitors provides insights into market trends and allows operators to benchmark against industry standards. This helps them stay competitive and differentiate their services based on performance.

3.7 Mobile User Interface Design

In our Android Studio project, the user interface (UI) components are organized into fragments and view models, each with specific responsibilities for handling data and user interface elements. The Home and Dashboard sections of the application each have corresponding fragment and view model classes to manage data flow and UI interactions.

The `HomeFragment` class serves as one of the two primary interfaces for presenting key information to the user. This fragment is where the queried network cell information listed in Section 3.1 are displayed, and it is designed to display them in real-time while ensuring interactive data experience. It also uses data binding to create a dynamic link between the user interface and the underlying data, allowing for automatic updates when the data changes. This class consists of several key elements:

- **Fragment Initialization**: The `onCreateView()` method is responsible for setting up the fragment's view. It inflates the layout and binds the

user interface element with data from the `HomeViewModel`, which allows for efficient updates without additional code.

- **LiveData Observers**: Various UI components, like `TextView`, are set up to observe changes in `HomeViewModel`'s `LiveData` objects. This ensures the UI updates automatically when underlying data changes. This reduces manual refreshes and ensures that the UI reflects the latest information.
- **Automatic Data Refreshing**: the `fieldsAutoRefresh` runnable updates the UI every 10 seconds. It uses the `Handler` to post delayed tasks, hence maintaining automatic refreshes and real-time updates without overburdening the system.
- **Cleanup**: The `onDestroyView()` method handles fragment cleanup when the view is destroyed. It sets the `binding` variable to `null` to release references to UI elements, avoiding memory leaks. It also removes the `fieldsAutoRefresh` task from the `Handler`, which stops automatic data refreshing to prevent unnecessary resource use.

The `HomeViewModel` class serves as the primary data management layer for the Home Fragment. As part of the Model-View-ViewModel (MVVM) architecture, this class is responsible for maintaining the state and business logic that drives the user interface. It operates as an intermediary between the fragment's user interface and the underlying data sources, ensuring a clean separation of concerns and promoting modular design. The `HomeViewModel` utilizes `MutableLiveData` objects to manage various aspects of the app's data. These objects allow the user interface to observe and react to changes in the data without explicit updates or callbacks.

- **Data Handling and Extraction**: The `updateAll()` method fetches real-time data from the `UIDataExtractor` class and updates the `MutableLiveData` objects, allowing the `HomeFragment` to reflect the latest changes in its view through data binding and `LiveData`. This triggers UI updates in the `HomeFragment` whenever underlying data changes.
- **Getters and Setters Methods**: These methods allow for external classes to retrieve or set values in the `MutableLiveData` objects. This provides a way to update the data without direct modification, ensuring a controlled flow of information.

The `DashboardFragment` is the second primary interface implemented in our Android application. It also aims to present statistical information to user, different from the ones presented in the `HomeFragment` class. This fragment is where the statistical information retrieved from the server are displayed (see Section 3.3). Similar to the first fragment, this one allows users to interact with various data points, set specific parameters (start date, end date), and view statistical information. It employs advanced features such as data binding and `LiveData` observation to ensure the UI remains responsive and up-to-date. This class also consists of several key elements:

- **Fragment Initialization**: Similar to `HomeFragment`, the `onCreate- View` method is where the fragment's view is constructed. It uses data binding to inflate the layout and connects the UI elements with the `DashboardViewModel`.
- **Date and Time Selection**: The fragment has functionality for users to select dates and times for data analysis and corresponding statistical retrieval. This is implemented through the `showDateTimeDialog()` method, which uses `DatePickerDialog` and `TimePickerDialog` to allow users to set exact period intervals.
- **LiveData Observers**: Similar to `HomeFragment`, this fragment sets up observers to update the UI whenever data changes in `DashboardViewModel`.
- **Cleanup**: The `onDestroyView()` method releases resources, ensuring proper cleanup when the fragment's view is destroyed.

Lastly, the `DashboardViewModel` class plays a crucial role in managing the data that drives the `DashboardFragment` in the Android application. It is responsible for storing and updating various key metrics that the fragment displays.

- **MutableLiveData Variables**: The `DashboardViewModel` class has several `MutableLiveData` objects to store various data fields related to the dashboard. This includes start and end dates, as well as the retrieved statistics from the server.
- **Getters and Setters Methods**: Similar to `HomeViewModel`, these methods allow other components to observe or update the data fields. They are crucial for ensuring the correct flow of information between the view model and UI components.

In short, these fragments and view models work together to separate concerns within the application. The fragments handle the user interface and trigger data updates, while the view models manage the underlying data and business logic. This organization supports a clean structure and improved maintainability in the Android app. The `HomeFragment` and `DashboardFragment` provide the user interface, while the `HomeViewModel` and `DashboardViewModel` manage data and logic.

4 Testing and Results

Testing is an essential step for any software project no matter the size or complexity. Hence, we spent a reasonable amount of time ensuring that all methods, tools, and components function seamlessly throughout the entirety of the application usage. The testing phase was completed over three stages, namely: unit and components testing, user interface testing, and load testing. These are detailed in the below sections.

- **Unit and Component Testing:** For each method created, we rigorously executed tests to identify runtime errors and ensure that outputs matched expectations. In addition, multiple tests were performed to ensure interactions between units and the overall performance of components. We carefully considered edge cases, encompassing all potential user interactions, to guarantee compatibility and functionality across various situations. This includes scenarios like users choosing one date instead of two for statistics generation, refreshing the page too often, or even connecting and disconnecting at an accelerated rate.

- **User Interface Testing:** To ensure optimal user experience, we meticulously tested the app's interface on diverse devices such as Pixel 8, Pixel Fold, and more using the Android Studio emulator. We also tested the application on physical Samsung devices such as Note20 and A52 models. We prioritized readability and clarity, ensuring that the interface remains intuitive and accessible across different screen sizes, resolutions, and displays (light mode vs dark mode). We conducted exhaustive testing to ensure seamless interaction between fragments. For instance, we had to deal with scenarios where the home fragment disappeared upon navigating to the dashboard. This thorough testing phase ensured that all fragments seamlessly communicated and updated data in real time, even when running in the background.

- **Load and Compatibility Testing:** To assess the app's performance under varying loads, we extensively connected and disconnected multiple devices simultaneously. Due to our limited amount of devices, we were only able to perform this task using two physical phones and one emulator. Nonetheless, we were still successful in monitoring the server's load adaptability and ensuring that no client was left unserviced during periods of high demand. We also conducted tests on devices with different API levels including two Android 14 devices, one Android 13 device, and varying Android versions on the emulator to ascertain the app's compatibility with newer devices.

By carefully conducting these tests across different aspects of the application, we aimed to deliver a robust, user-friendly, and reliable product that excels in performance and functionality across diverse user scenarios.

To visualize the overall functionality and results of our mobile application, we conducted a sample run that simulated a typical client's usage of the app and server. We started by running the flask server before connecting our mobile device by opening the app. The connection was almost instantaneous and within ten seconds the server was receiving all the required data from our mobile device. We went on to check that our data was being entered properly into the database before requesting statistics from the server and ensured that all communications were running smoothly. As for the server interface, we were able

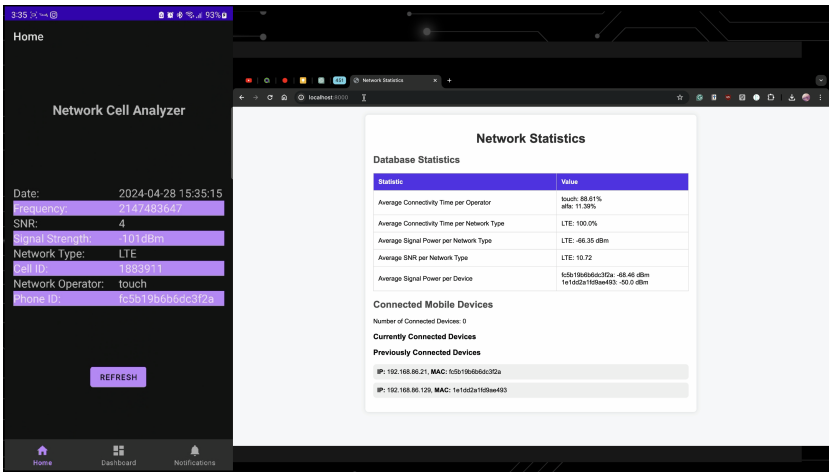

Figure 3: Network Cell Analyzer Application and Server Sample Run

to display certain statistics and all previously connected devices, however, dynamically updating the currently connected devices was posing some issues. Even after debugging for multiple hours and consulting outside resources, the problem was not resolved and we suspect that it is due to the nature of our socket connection as the client disconnects after every send/receive operation thus counting is not practical in our case. Figure 3 displays a screenshot from our demonstration, showcasing the server interface as well as the home page of our application.

5 Conclusion

The Network Cell Analyzer offers a robust architecture for real-time monitoring and analysis of cellular network data, combining Android applications with Python-based servers to deliver a scalable system. The app can extract, process, and present crucial network statistics in a user-friendly interface, providing network operators with valuable insights into connectivity, signal power, and SNR for informed decision-making.

Our approach employs modern software engineering principles such as MVVM architecture and LiveData observation, ensuring a responsive user experience. Server-side processing with Flask and SQLAlchemy facilitates efficient data handling and supports scalable query operations. Extensive testing, including component, UI, and load compatibility testing, guarantees stability and compatibility across different devices.

Despite these successes, there are areas for continued improvement. Further research is needed to enhance resource allocation, data visualization, and data validation processes. Keeping up with evolving technologies and user demands will be crucial for maintaining the application's relevance and effectiveness in

the field of mobile network analysis.
In summary, our project began by establishing a robust architecture, focusing on mobile data extraction and server-side processing. We explored the key functionalities, from querying and visualizing network statistics to seamless UI updates. As we move forward, we recognize the importance of ongoing research and development to address current challenges and ensure the long-term success of the Network Cell Analyzer.

References

[1] Yuanjie Li et al. "Mobileinsight: extracting and analyzing cellular network information on smartphones". English. In: ACM, Oct. 2016, pp. 202–215. ISBN: 9781450342261;1450342264;

[2] Android Developers. "Settings.Secure". In: 2024. URL: `https://developer.android.com/reference/android/provider/Settings.Secure`.