SAPIENZA
UNIVERSITÀ DI ROMA

Implementation of a Wi-Fi Sensor for the GeneroCity App on Android

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Applied Computer Science and Artificial Intelligence

Simone Russolillo
ID number 1985206

Advisor
Prof. Emanuele Panizzi

Academic Year 2023/2024

Thesis not yet defended

Implementation of a Wi-Fi Sensor for the GeneroCity App on Android
Bachelor's degree internship report. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: simonerussolillo02@gmail.com

Contents

List of Figures

Chapter 1

Introduction

1.1 The Growing Parking Dilemma

As urban centers expand at an unprecedented rate, the demand for parking availability has become a critical issue. With vehicle ownership on the rise, this surge in cars is creating significant challenges for city residents.

In fact, it is estimated that around 30% of urban traffic is generated by drivers merely searching for parking spots, contributing to the overall congestion in major cities like New York, Tokyo, and Paris.[5] Beyond traffic, the environmental and social costs are profound—ranging from increased air pollution and wasted time for motorists to heightened stress levels.

Recognizing these problems, cities like Amsterdam and Copenhagen are rethinking urban planning to reduce car dependency by enhancing public transport, promoting cycling infrastructure, and encouraging the development of pedestrian-friendly spaces.

However, transitioning to these solutions will not happen overnight. Large-scale changes, particularly those that involve redesigning infrastructure and shifting cultural habits around car usage, are complex and time-consuming.

The city of Rome, for instance, is no exception to this global trend. With its historical layout and narrow streets, parking issues are especially challenging. Although efforts have been made to promote alternative modes of transportation, Rome continues to grapple with traffic congestion, high vehicle emissions, and the ongoing search for parking, just like many other major urban hubs.

1.2 GeneroCity: An App for Smart Parking

While long-term urban planning initiatives are underway, immediate solutions have emerged to tackle the growing parking crisis. One such innovation is *GeneroCity*, a smart parking application designed for both Android and iOS.[4] Developed by the Gamification Lab at the Department of Computer Science, University of Rome "La Sapienza", this app offers a creative approach to reducing parking stress in congested urban areas.

The app works by promoting a sense of community, encouraging users to share available parking spaces with one another. By leveraging the principle of generosity, *GeneroCity* aims to ease the exchange of parking spots in real time, making it easier for drivers to find spaces without endlessly circling city streets.

Figure 1.1 GeneroCity Logo

In addition to its core functionality, *GeneroCity* offers features that enhance convenience. Users can manage vehicle information directly within the app, even sharing it with family members to coordinate parking more effectively. What's more, the app is designed with simplicity in mind—no registration process is required. Upon download, users simply grant access to location services and notifications, allowing them to begin using it immediately.

This practical and user-friendly solution stands out as a valuable tool for alleviating parking-related frustrations, especially in cities like Rome where the need for efficient parking management is urgent.

1.3 How GeneroCity Works

GeneroCity enhances urban parking by allowing users to communicate when they vacate a parking spot, creating a real-time exchange that benefits everyone searching in the vicinity.
To notify others of an available space, users simply press the **LEAVE PLACE** button. Meanwhile, those in search of parking can easily tap the **SEARCH PLACE** option to discover openings nearby.
When users launch the app for the first time, they are greeted with a preconfigured vehicle profile, ready for immediate use. This initial setup can be tailored by adding essential details such as license plate, name, model, and even a nickname for personalization.
Moreover, *GeneroCity* supports multiple vehicles, allowing users to register several cars within the app. This flexibility ensures that whether you're using your family car, or your own vehicle, the app remains an invaluable tool for navigating parking in bustling urban environments.

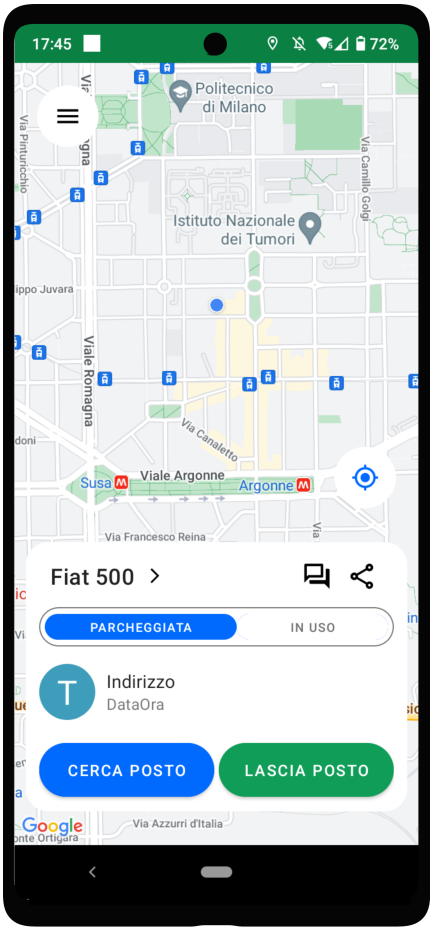


Figure 1.2 Android App Screen

1.3.1 Keeping Track: The Parking Activity Log

GeneroCity offers users a comprehensive Parking Activity Log for each registered vehicle, enabling them to keep track of their parking history with ease. This log provides valuable insights into past parking activities, including:

- **Start date and time** of parking
- **End date and time** of parking
- **Type of parking** (e.g. angled parking, parallel parking, etc.)
- **Street location** where the vehicle was parked

By maintaining a detailed record of parking instances, users can better understand their habits, optimize their parking strategies, and make informed decisions for future parking needs. This feature enhances the overall user experience, making *GeneroCity* not just a tool for finding parking, but also a resource for smarter urban navigation.

1.3.2 Automating the Parking Process for Seamless Use

To elevate the user experience and reduce distractions while driving, *GeneroCity* incorporates features designed to automate the parking process. These advanced capabilities aim to predict when a user is about to search for or vacate a parking spot, streamlining the interaction with the app.

The system utilizes **programmed sensors** that assess the user's status in real time, ensuring that notifications and functions are activated only when needed. This proactive approach not only simplifies the parking experience but also enhances safety by minimizing the need for users to interact with their phones while on the road.

Further exploration of this innovative concept will be provided later, showcasing how technology can transform urban parking into a more efficient and user-friendly experience.

Chapter 2

The Architecture

GeneroCity is an innovative application that utilizes a variety of sensors to collect information about users' status and activities, ultimately enhancing the parking experience. While the vision includes around twenty different sensors, we have designed and implemented three key components so far.

These sensors are integral to the overall functionality of *GeneroCity*, enabling effective data collection and interaction. Each implemented sensor plays a critical role in optimizing user experience by ensuring the app responds to real-time conditions and needs. This foundational work paves the way for future expansions and enhancements to the sensor capabilities within the application.

2.1 Central System and Data Logging

The structure of the GeneroCity application can be envisioned as a central system that acts as an intermediary between the database and the various sensors, coordinating their interactions.

When a sensor updates its confidence level, it notifies the central body, which then queries all other sensors to retrieve their confidence levels at that specific moment. This allows the application to accurately assess the user's status based on input from multiple sensors.

Additionally, the central system automatically logs sensor data into GeneroCity's database, ensuring that all relevant information is stored and available for further analysis. Let's delve deeper into the details of the central system starting from the Sensor Interface.

2.2 The Sensor Interface

In the GeneroCity app for Android, all sensors implement the **GCSensorInterface**. This interface standardizes sensor behavior by providing common methods and attributes, streamlining the development and ensuring a consistent architecture across all sensors. One of the most critical methods in this interface is getStatus.

```
public double getStatus(Calendar timestamp) {
    Map.Entry<Long, Double> closestStatus = closestStatusTo(timestamp);
    return closestStatus == null ? 0.5d : closestStatus.getValue();
}
```

The **getStatus** method allows the app to query a sensor's state based on a specific timestamp. It returns a value between 0 and 1, known as the "**Confidence**", which reflects the probability that the user is in a certain state:

- **A value between 0 and 0.5** suggests that the user is likely in "Walking" mode, with 0 representing complete certainty.
- **A value of exactly 0.5** indicates uncertainty, where the sensor has no clear indication of the user's state.
- **A value between 0.5 and 1** points to the likelihood that the user is in "Automotive" mode, with 1 indicating complete certainty.

Another important method is **update**.

```
public void update(Calendar timestamp, SensorData sensorData) {
    collect(timestamp, sensorData);
    GCSensorConstants.onUpdate(timestamp);
}
```

This function is called whenever a sensor changes its confidence level for a specific timestamp. The update method serves two critical roles:

- It automatically logs the sensor data into GeneroCity's database for future analysis and reference.
- It triggers a recalculation for all sensors, ensuring that the entire system is updated and synchronized based on the latest sensor data.

This architecture ensures that each sensor in the GeneroCity system operates autonomously, updating its confidence level independently while contributing to a cohesive, system-wide recalibration when necessary. This design makes it easier to implement new sensors while maintaining a stable and consistent platform.

2.3 The Sensor Constants Class

The **GCSensorConstants** class acts as a utility that manages the collection of sensors within the GeneroCity app, coordinating their updates to determine the user's current movement state. It maintains a historical record of sensor data, calculates new states, and handles corresponding actions based on sensor readings.

The core method of this class is **onUpdate**, which is triggered by the sensors when a state change occurs. This method ensures that updates are processed in an orderly manner, preventing concurrent updates from disrupting the system. When an update is triggered, the **compute** function is called.

```
private static double compute(Calendar time) {
    double sum = 0.0;
    double wei = 0.0;

    double confidence;
    for (GCSensorInterface module : sensors) {
        confidence = module.getStatus(time);

        sum += (confidence - 0.5d) * module.weight;
        wei += module.weight;
    }

    return sum / wei + 0.5d;
}
```

This function gathers data from all active sensors, calculating a weighted average of their confidence values to determine the user's current state: "Walking" or "Automotive" mode.

2.4 Integrated Operation of Class and Interface

The GCSensorConstants class serves as the central coordinator, integrating data from sensors that implement the GCSensorInterface. The process works as follows:

1. When a sensor detects a change in the user's state, it invokes the **update** method. This action logs the new data and signals the central system to process the update.
2. The **compute** function is then called to analyze the change. It queries all sensors using the **getStatus** method to gather their current confidence values and calculates the overall state.
3. Based on the computed state, the application determines the appropriate action, such as whether to indicate a parking space is available or not.

2.5 List and Functionality of Sensors

Below is a description of the currently implemented sensors and their functionality.

- **BatterySensor**: This sensor determines the user's state based on the phone's charging status and battery voltage. If the phone is connected to a charger, it suggests a higher likelihood that the user is in a vehicle, while being unplugged lowers that likelihood. Additionally, a higher voltage indicates that the user is less likely to be in a car, whereas a lower voltage suggests the opposite.
- **CellSensor**: This sensor analyzes mobile network data, including signal strength, cell ID, and location coordinates, to determine the user's current state. By evaluating changes in these attributes, it assesses whether the user is stationary or moving.

- **BluetoothSensor**: This sensor detects connections to Bluetooth devices, such as car stereos or Android Auto, to assess whether the user is in a vehicle. By monitoring active Bluetooth connections, it can identify if the phone is paired with automotive systems, providing a strong indicator of the user's current mode of transportation.

Chapter 3

Wi-Fi Sensor

The Wi-Fi sensor is designed to detect "fixed" access points that the user frequently connects to, such as at home, the gym, or work. By identifying these consistent connections, the sensor can learn the user's routine and habits, allowing it to infer whether the user is likely stationary or in transit. This insight helps determine if the user is inside a vehicle or in a familiar, fixed location, contributing to a more accurate assessment of their activity.

3.1 Setting Up the Wi-Fi Sensor

3.1.1 App Permissions

The first step in implementing the Wi-Fi sensor was ensuring that the app had the necessary permissions to access the device's Wi-Fi state and location data.[2] This involved adding the following permissions to the **AndroidManifest.xml** file:

- `ACCESS_WIFI_STATE` – to allow the app to detect nearby Wi-Fi networks and determine the current connection status.
- `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` – to grant access to location services, necessary for retrieving accurate latitude and longitude data.

While Wi-Fi access is essential for detecting connections, the location permissions play a key role in improving the sensor's ability to make more informed assumptions about the user's state, as will be explained further in the decision making process.

3.1.2 Android Services and Broadcast Receivers

In Android, services play a vital role in enabling background processing.[3] These components allow apps to perform long-running operations or react to system events without a user interface. Services are essential for tasks such as monitoring sensor data, fetching remote information, or handling device status changes (like network status).

Android uses several system services to interact with device hardware, and apps can register broadcast receivers to listen for specific system broadcasts, such as changes in battery state, network status, or Wi-Fi connection.

3.1.3 At a glance

- **System Services**: These are essential system-level components that allow apps to interact with the device's hardware, such as the WifiManager in this case, which provides detailed information about the Wi-Fi.
- **Broadcast Receivers**: A lightweight component that allows an app to listen for and react to system-wide broadcast messages. These broadcasts are often triggered by changes in the system or hardware, like changes in battery level or network connectivity.

3.2 First Version of the App

Let's explore the technical details of how I implemented the Wi-Fi sensor in the first version of the app.

3.2.1 Structure of the Wi-Fi Sensor Implementation

In the GeneroCity app, the Wi-Fi sensor is implemented to monitor and react to changes in the device's connection status. The sensor uses Android's WifiManager system service and a custom broadcast receiver to gather and process Wi-Fi-related data.

WifiSensor.java

- **Purpose:** This class initializes the Wi-Fi sensor and registers it with Android's broadcast system to receive updates on connectivity changes.
- **WifiManager:** The WifiManager is obtained from the system to access detailed Wi-Fi information.
- **Broadcast Receiver:** The app creates an instance of the WifiBroadcastReceiver and registers it using an IntentFilter. The receiver is designed to listen for Wi-Fi-related events, specifically NETWORK_STATE_CHANGED_ACTION.

WifiBroadcastReceiver.java

- **Purpose:** This class extends Android's BroadcastReceiver and is responsible for reacting to the Wi-Fi change broadcasts.
- **onReceive Method:** The key method in this class is onReceive, which is triggered every time a relevant broadcast is received (such as when a connection is initiated). It uses the WifiManager to extract various pieces of information about the Wi-Fi connection, such as the SSID, signal strength, and connection status.
- **Data Processing:** After gathering the Wi-Fi data, the broadcast receiver sends this information back to the WifiSensor class through the onData method. This allows the sensor to log the data or trigger further actions based on the Wi-Fi state.

General Flow

- **System Broadcast:** When the Wi-Fi state changes (e.g., connection to new access point), Android sends out a system-wide broadcast.
- **Broadcast Receiver:** The WifiBroadcastReceiver listens for this broadcast, receives the message, and collects detailed battery information from the WifiManager.
- **Data Processing:** The collected data is passed to the WifiSensor, which handles it (e.g., logging or making decisions based on the data).

This implementation structure leverages system services to access hardware data and broadcast receivers to react to system events. It provides an efficient way to monitor and respond to Wi-Fi-related changes without needing constant user interaction, a common design pattern in Android applications for handling background tasks and hardware monitoring.

3.2.2 Data Logging: WifiListActivity

At this stage, the app can detect whether the device is connected to a network, with the primary focus on capturing the SSID of the connected network. To achieve this, I developed a basic textual user interface, within an activity called **WifiListActivity**.

But first, what is an Android activity?

Android Activities

An activity in Android is a fundamental component that provides a user interface for interaction, allowing users to engage with the app's features and functionalities.[1] Each activity is implemented as a subclass of the 'Activity' class and typically contains UI elements such as buttons, text fields, and images arranged within a layout.

Activities can also bind to services, enabling them to communicate and exchange data. This binding allows an activity to receive real-time updates from a service running in the background.

In this case, **WifiListActivity** logs and displays the intervals of connection and disconnection for each network. Although this is a temporary solution, it provided a clear representation of the app's functionality and helped clarify the next steps in development. An example of the app's display is also available.

3.3 Second Version of the App

In the second version of the app, I was tasked with significantly enhancing the user experience. To achieve this, I decided to replace the basic textual interface with a dynamic map, which aimed to provide a clearer visual representation of the data. Here are the key implementations made in this version:

- **Dynamic Map Interface**: The static text display was transformed into an interactive map, allowing for a more engaging user experience.
- **Annotations for Wi-Fi Connections**: Each Wi-Fi connection event is marked on the map with annotations, showing the exact coordinates where the connection occurred.
- **Detailed Connection Periods**: For each access point, the app now logs detailed connection intervals, enabling users to see when they connected and for how long they stayed connected to a specific Wi-Fi network.

To achieve this, I added an extra activity, which introduced some complexity to the app's structure. While these enhancements improved usability and data presentation, managing the new data flow required careful consideration.

3.3.1 Structural Changes in App Design

By adding more activities that required the same data, I encountered the issue of duplicate signals caused by multiple instantiations of 'WifiManager' and 'WifiBroadcastReceiver' for the same service. Each instance could independently listen for updates and broadcast events, leading to redundant data processing and potentially inconsistent behavior within the app.

To address this, I decided to consolidate these components into a single instance, ensuring that all parts of the application interact with a synchronized, shared broadcast receiver. This approach eliminates redundant signals and ensures a more efficient handling of Wi-Fi data.

As demonstrated in the code snippet above, the 'WifiBroadcastReceiver' is instantiated only once. When an activity needs Wi-Fi data, it retrieves that single instance and adds itself to the list of listeners managed by the 'WifiBroadcastReceiver' class, ensuring consistent and streamlined data flow across all activities.

Key Aspects of Synchronization and Listener Management

1. **Thread Safety**: The 'listeners' list is wrapped in synchronized methods, ensuring that modifications to the list—such as adding or removing listeners—are safe from concurrent access issues.

2. **Efficient Listener Notification**: The 'notifyListeners' method iterates through the 'listeners' array and calls the 'onData' method on each registered listener.

3. **Dynamic Listener Management**: The 'addListener' and 'removeListener' methods allow for dynamic management of listeners. When a new component needs to listen for updates, it can simply register itself, and when it no longer requires updates, it can deregister.

4. **Avoiding Multiple Instances**: Instead of creating separate instances of 'WifiBroadcastReceiver' in every activity or service that requires Wi-Fi data, using a singleton instance helps in consolidating all listeners into one location. This prevents unnecessary overhead and complexity associated with managing multiple instances.

Overall, this design choice not only simplifies data flow within the app but also enhances performance and reliability by ensuring that all parts of the application can respond to Wi-Fi updates consistently and efficiently.

3.3.2 Data Logging: WifiMapActivity

Going back to the implementation of our map, the focus is on the WifiMapActivity, which leverages Google Maps to display markers at the geographic locations of each Wi-Fi connection event.

The app uses Android's location services to obtain the user's location and display it on the map. When the map is ready, the app retrieves previously recorded Wi-Fi data through the 'WifiBroadcastReceiver', which listens for Wi-Fi events and provides a list of Wi-Fi connections with their associated geographic coordinates.

Map Markers and Bottom Sheets

For each detected Wi-Fi connection, a marker is placed on the map at the corresponding location. When a user clicks on a marker, the app centers the map on that location and opens a bottom sheet, implemented via the 'BottomSheetWifiMarker' class.

This bottom sheet shows the SSID of the network and the connection intervals, which indicate when the device was connected to or disconnected from that network. The connection intervals are displayed in a readable format using a 'SimpleDateFormat', which formats the timestamps into a string that includes both the start and end times of the connection.

By using this map-based interface, users can visually track their Wi-Fi connections over time and space, enhancing the clarity and functionality of the app. Each marker represents a distinct access point, and the map itself provides a geographic context to better understand user movement and Wi-Fi network interactions.

3.3.3 The Graphical Interface

In the left image (Figure 4.1), the app's graphical interface displays a map with markers indicating the geographic locations of past Wi-Fi connections, along with the corresponding SSIDs for each network. Each marker represents a unique access point that the device connected to.

In the right image (Figure 4.2), tapping on a marker brings up detailed information for that network. In this example, the marker for the network "FurgoneDellaDigos" is selected, revealing a list of connection intervals in a bottom sheet. This interface allows users to visually track their connection history and examine the precise times they were connected to specific Wi-Fi networks.

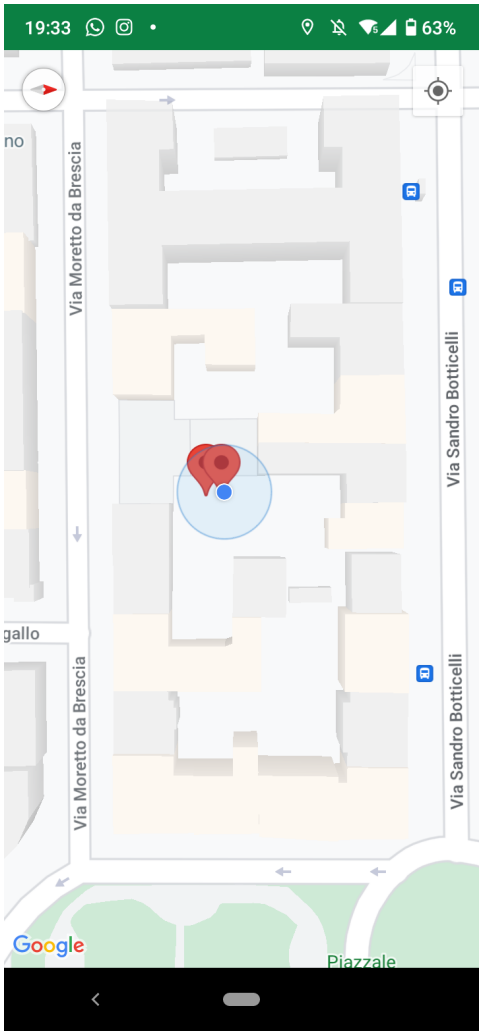


Figure 3.1 App interface with markers on network coordinates

Figure 3.2 Result of tapping on a marker

3.3.4 Next Steps in Development

At this stage of development, two essential features were identified to turn the app into a fully functional sensor:

- **User State Calculation**: This feature would calculate the user's status, providing a value between 0 and 1 to determine the likelihood of the user being in motion or stationary.
- **Fixed vs. Mobile Access Point Detection**: This feature would distinguish between fixed networks (e.g., home or office) and mobile networks (e.g., temporary hotspots), enhancing the app's ability to categorize network types.

These functions would be integrated into the Wi-Fi sensor within the Generocity app.

Chapter 4

Sensor Implementation on GeneroCity

4.1 Initial Decisions

The confidence range, which indicates whether the user is in a "walking" or "automotive" state at a given time, has been adjusted from 1-0 to 0.5-0. This decision was made because, based solely on Wi-Fi information, it is only possible to have confidence within the 0-0.5 range. This range reflects the probability that the user is in "walking" mode, with 0 representing absolute certainty.

Before explaining how the confidence calculation and the detection of fixed access points were implemented, it's important to review the changes made to the code and the structure of the new 'WifiSensor' class.

4.2 The Battery Issue

Generocity prioritizes the energy efficiency of the device. One of its functionalities is the ability to set a minimum distance (measured in meters) that the device must travel before an update event can be triggered by the LocationRequest.

This is achieved through the 'distanceFilter' property, which in Generocity is set to 25 meters. By implementing this filter, the application can significantly reduce battery consumption. However, this comes with a trade-off: a slight decrease in precision when retrieving data from the 'WifiSensor'.

```
// LOCATION UPDATES
private void startLocationUpdates(Context context) {
    LocationRequest locationRequest = LocationRequest.create()
            .setInterval(10000)  // 10 seconds interval
            .setSmallestDisplacement(25);  // 25 meters

    locationCallback = new LocationCallback() {
        @Override
        public void onLocationResult(@NonNull LocationResult locationResult) {
            for (Location location : locationResult.getLocations()) {
                onLocationChanged(location);
            }
        }
```

```
    };

    if (ActivityCompat.checkSelfPermission(context, Manifest.permission.
    ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED && ActivityCompat.
    checkSelfPermission(context, Manifest.permission.ACCESS_COARSE_LOCATION) !=
    PackageManager.PERMISSION_GRANTED) {
        return;
    }
    fusedLocationClient.requestLocationUpdates(locationRequest, locationCallback,
    Looper.getMainLooper());
}
```

The code snippet demonstrates how location updates are initiated, specifying a 10-second interval and a displacement of 25 meters, thereby balancing the need for accurate location tracking with battery conservation.

4.3 Access Point

The ‘WifiData‘ class is a foundational component of the app, designed to represent each Wi-Fi access point the device interacts with. It encapsulates essential data such as the network’s SSID, geographic coordinates, and connection intervals. This structure is vital for efficiently logging, organizing, and retrieving Wi-Fi information throughout the app. Below is a breakdown of its key attributes and methods:

Attributes

```
// CONSTRUCTOR
public WifiData(String ssid, double latitude, double longitude) {
    this.id = UUID.randomUUID();
    this.ssid = ssid;
    this.latitude = latitude;
    this.longitude = longitude;
    this.isHome = false;
    this.connectionIntervals = new ArrayList<>();
}
```

- **id**: This ‘UUID‘ uniquely identifies each access point. It is automatically generated when an access point is created, ensuring that even networks with the same SSID are tracked independently.
- **ssid**: This ‘String‘ holds the name of the Wi-Fi network (SSID), allowing the app to differentiate between the networks the device connects to.
- **latitude & longitude**: These ‘double‘ fields store the geographical coordinates (latitude and longitude) of the access point at the time of connection. This information is crucial for mapping Wi-Fi locations and tracking movements in relation to network connections.
- **isHome**: This ‘boolean‘ flag indicates whether the network is classified as a "home" network. Initially set to ‘false‘, this value is updated later based on certain criteria that we will discuss in detail.

- **connectionIntervals**: This ‘List‘ stores arrays of ‘Date‘ objects, each representing the start and end times of a connection to the access point. While connected, the end time is set to a distant future date, later updated when the device disconnects.

Methods

- **addNewConnectionInterval()**: This method creates a new connection interval every time the device connects to an access point, marking the current time as the start. It helps log connection periods efficiently.
- **setEndDate()**: When the device disconnects from a Wi-Fi network, this method updates the most recent connection interval, setting the end date to the time of disconnection. This ensures that each connection period is fully recorded.
- **calculateDisplacementFrom(lat, lon)**: This method calculates the distance between the stored coordinates of the access point and a given latitude and longitude. It is useful for determining how far the user has moved from the Wi-Fi network's location.

```
public double calculateDisplacementFrom(double lat, double lon) {
    double lat1Rad = Math.toRadians(this.latitude);
    double lat2Rad = Math.toRadians(lat);
    double lon1Rad = Math.toRadians(this.longitude);
    double lon2Rad = Math.toRadians(lon);

    double x = (lon2Rad - lon1Rad) * Math.cos((lat1Rad + lat2Rad) / 2);
    double y = lat2Rad - lat1Rad;

    return Math.sqrt(x * x + y * y) * EARTH_RADIUS;
}
```

There are other methods, such as those related to determining whether a network is "home" or resetting Wi-Fi data, which we will discuss in detail later. Overall, the ‘WifiData‘ class is crucial for capturing and managing Wi-Fi data, enabling the app to log Wi-Fi connections and display their relevant details accurately.

4.3.1 CheckIfHome

In this section, we delve into the mechanism that allows the app to distinguish between "fixed" access points—such as home, work, or gym networks—and "non-fixed" access points like mobile hotspots or public Wi-Fi networks.

We also explore how the two functions, **checkIfHome** and **resetWifiData**, work together to ensure this distinction. The former determines whether an access point qualifies as "fixed," while the latter resets the data if the network moves to a significantly different location.

The ‘checkIfHome‘ function uses a simple condition: if the number of connection intervals exceeds 10, the network is classified as "fixed." In such cases, the ‘isHome‘ variable for that access point is set to ‘true‘.

Here’s the code for ‘checkIfHome‘:

```
public void checkIfHome() {
    this.isHome = this.connectionIntervals.size() > 10;
}
```

Interaction Between ‘checkIfHome‘ and ‘resetWifiData‘

Whenever the location of a known Wi-Fi network is detected to have shifted significantly (over 100 meters), the ‘resetWifiData‘ function is called. This function clears all previous connection data and resets the coordinates of the access point. Consequently, when the ‘checkIfHome‘ function runs again, the network is re-evaluated and no longer considered "fixed" due to the reset.

Here’s an overview of ‘resetWifiData‘:

```
public void resetWifiData(double lat, double lon) {
    // id =  stays the same
    // ssid = stays the same
    this.latitude = lat;
    this.longitude = lon;
    this.isHome = false;
    this.connectionIntervals = new ArrayList<>();
    addNewConnectionInterval();
}
```

By coupling these two functions, the app can reliably differentiate between a stable "home" network and one that is transient. If a Wi-Fi access point consistently has more than 10 connections and hasn’t been reset due to location changes, it’s deemed to be a fixed network. On the other hand, if a location shift is detected, the ‘resetWifiData‘ method clears the connection history, requiring the app to "re-learn" whether the network should be treated as a home network.

This mechanism ensures the accuracy of location-based Wi-Fi data logging and prevents networks that are mobile or otherwise temporary from being treated as home or workplace networks.

4.3.2 Wi-Fi Event Processing

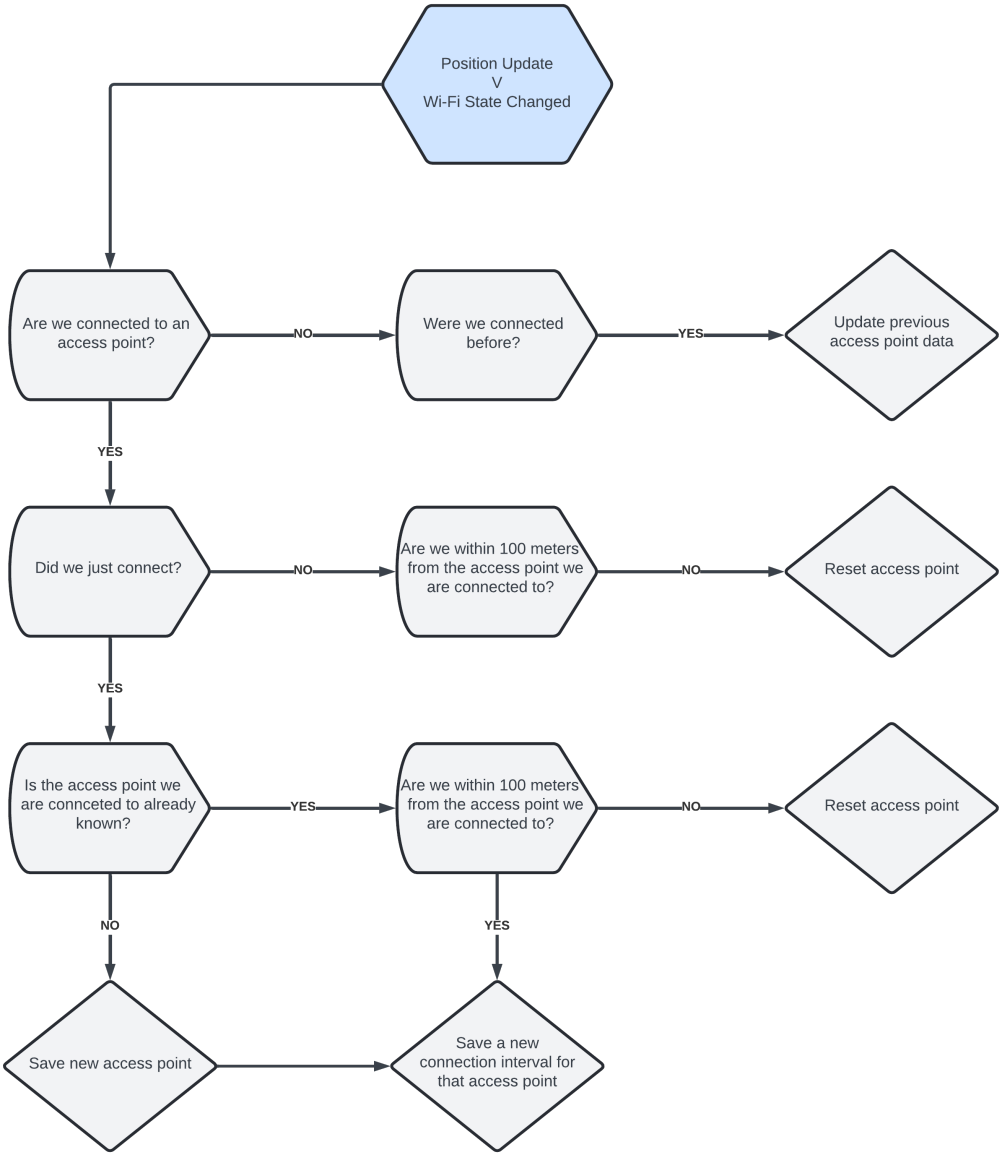
Let's break down the flow of the **onReceive** method in the WifiBroadcastReceiver, highlighting what happens in each case and the relevant method names.

1. **Opening the App**: When the app is launched, it first checks for the presence of an SSID.
   - **No SSID Detected**:
     - **Disconnection**: If the SSID is unknown ssid, this indicates a disconnection. The method setEndDate() is called on the first entry in the listOfWifi to mark the disconnection, followed by notifyListeners() to inform other components of this change.
     - **Wi-Fi Off at Launch**: If the SSID is unknown, the app recognizes that Wi-Fi may have been turned off at the start.
2. **Connection to Wi-Fi**: If an SSID is detected and differs from the last recorded SSID, the app processes the connection.
   - **New Access Point**: The app checks if the SSID exists in the current list of connections. If it does not:
     - It retrieves the current location. If the location is available, it creates a new WifiData object for this access point, capturing the SSID and location coordinates. The method addNewWifiToList(newWifi) is called to add it to the list, followed by notifyListeners() to inform listeners of this new connection.
   - **Known Access Point**: If the SSID matches an existing entry:
     - The app retrieves the corresponding WifiData object, calls addNewConnectionInterval() to add a new connection interval, and checkIfHome() to determine if the access point qualifies as "home". It then retrieves the current location again, and if available, calculates the displacement.
     - If the displacement exceeds 100 meters, the app calls resetWifiData(latitude, longitude) to update the WifiData with the new location. The existing entry is then updated in the list, followed by notifyListeners() to notify of these changes.

This structured approach, utilizing specific methods for each task, allows the app to effectively track and manage Wi-Fi connections, ensuring accurate updates and user notifications.

4.3.3 Flow Diagram

The following page presents a flow diagram that summarizes the process of detecting and storing sensor information. This diagram visually outlines the key steps involved in how the sensors operate within GeneroCity, providing a clear representation of the data flow and the interactions between various components of the system.



I would like to extend my heartfelt gratitude to **Riccardo Calderoni** for his invaluable contributions to the design of the flow diagram featured in this section. His thoughtful approach to the implementation details of GeneroCity for the iOS platform has been instrumental in shaping the application's functionality.

4.4 Saving Sensor Data

Initially, the app stored Wi-Fi data in a local variable. While this allowed for capturing and displaying data within the app, all data was lost once the app was closed, as no persistent storage was implemented. This was intended as a temporary setup during the early development phase to observe and handle the captured data without complications related to storage.

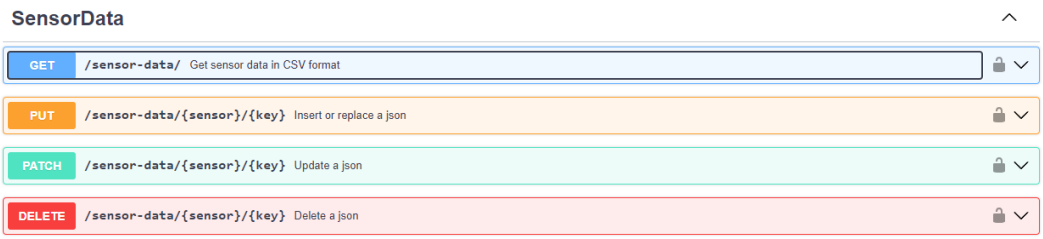
To address the need for persistent data storage, I implemented a method to save sensor data to a database. The key improvement was incorporating the following logic:

```
boolean isChanged = lastConfidence != (lastConfidence = currentConfidence);
if (isChanged) {
    Calendar calendar = Calendar.getInstance();
    sensorHistory.put(calendar.getTimeInMillis(), lastConfidence);
    update(calendar, new SensorData(calendar, lastConfidence, listOfWifi.get(0)));
}
```

This code performs several critical tasks:

- It checks if any data has changed, indicating a need to update the stored records.
- A Calendar instance captures the current time, which is used as a timestamp.
- The current confidence level is saved along with the relevant Wi-Fi data by invoking the update() method.

This enabled communication with the backend server by using a RESTful API, facilitating the storage of Wi-Fi and sensor data on a central database. The server exposed several endpoints for managing sensor data, such as:



These endpoints allow the app to not only store Wi-Fi and sensor data but also query, update, or delete records as needed, ensuring that the app can retain user data across sessions and provide a more robust and scalable experience.

4.5 getStatus

The final step in processing the sensor data is determining the user's location confidence level, which is based on the Wi-Fi data and the device's geographic location.

The method responsible for this logic is **onData()**. This method first retrieves the current SSID from the Wi-Fi manager and checks the device's location permissions. Once permission is granted, the app fetches the device's last known location using the FusedLocationProvider.

The decision-making process for calculating the user's location confidence is as follows:

1. **Home Wi-Fi Detected**: If the current SSID is recognized as the home Wi-Fi (determined through the getIsHome() method in the WifiData object), the app sets the confidence to **0.0**, indicating that the user is at home.

2. **No Wi-Fi or Unknown SSID**: If the current SSID is "<unknown ssid>" or if no Wi-Fi is detected, the app evaluates the user's displacement from previously known Wi-Fi locations:

    - If the user is within 100 meters of a saved Wi-Fi access point that is marked as home, the confidence is set to **0.3**, implying that the user is near home.
    - If the user is more than 100 meters away from any known Wi-Fi access points, the confidence is set to **0.5**, indicating that the user is far from home.

By analyzing both the current Wi-Fi connection and the user's proximity to saved access points, the app can effectively assess whether the user is at home, nearby, or far away. This confidence level plays a critical role in enhancing the accuracy of location-based features within the app, making it adaptive to changes in the user's surroundings.

Chapter 5

Conclusion

In conclusion, the GeneroCity project presents a promising solution to the persistent problem of traffic congestion, particularly in urban areas where the daily search for parking can be a frustrating experience for many. By leveraging technology to facilitate smarter parking choices, GeneroCity not only eases the burden on drivers but also contributes to a more efficient use of urban space.

Reflecting on my time in the lab, I realize that my journey was not just about acquiring technical skills; it was about applying those skills to address a significant real-world challenge. This aligns with the fundamental mission of computer science: to innovate and create solutions that enhance everyday life.

Looking ahead, I envision a future where GeneroCity evolves into a more sophisticated tool through the implementation of a machine learning model. By harnessing the wealth of data collected from various phone sensors, we can empower the application to make even more accurate and personalized parking recommendations. This vision not only has the potential to improve user experience but also to significantly reduce traffic congestion, paving the way for smarter, more sustainable urban environments.

As we continue to explore the intersections of technology and societal needs, I am excited about the role that GeneroCity could play in shaping a more convenient and efficient future for drivers everywhere.

Bibliography

[1] *androidActivities*. URL: https://developer.android.com/guide/components/activities/intro-activities.

[2] *androidPermissions*. URL: https://developer.android.com/guide/topics/permissions/overview.

[3] *androidServices*. URL: https://developer.android.com/develop/background-work/services.

[4] *generocity*. URL: https://www.generocity.it/.

[5] *parkingNetwork*. URL: https://www.parking.net/parking-news/skyline-parking-ag/traffic-congestion.