Queen's University

PEPPERMINT: WearOS Application – Benchmark for Exploiting Raw Sensor Data Access and Recording Capabilities

Tantakoun, Marcus / 20233273, Vanshita, Uthra / 20225327
CISC 499 Undergraduate Capstone Project
Supervisors: Professor Christian Muise, Professor Furcan Alaca

Winter Semester 2024

Background

With the rise in popularity of the rapidly evolving landscape of wearable technology, the integration of advanced sensor capabilities within smartwatches has opened up a plethora of opportunities for innovative applications. To conduct ongoing research in both continuous authentication [10] and smart office analytics, this project aims to build a custom Android Wear application that will let us use all the sensors in modern smartwatches. Students located in Kingston will optionally have access to modern wearables to test the developed application, and the resulting prototype will contribute directly to research in both the MuLab and CSRL research labs.

Motivation

The idea of this project stemmed from the growing demand for wearable applications that offer deeper insights into health, fitness, and environmental data, and how users can manipulate their data for further research. Traditional wearable applications often provide pre-processed sensor data, limiting users' ability to analyze and interpret raw data streams. Recognizing this gap, the development team embarked on creating *Peppermint*, a WearOS app that empowers users with direct access to raw sensor data, enabling advanced monitoring, analysis, and customization of their particular applications.

Key Objectives:

1. Enable the users to access raw sensor data: *Peppermint* prioritizes providing users with unfiltered and filtered access to all sensors embedded within their specific WearOS devices.
2. Facilitate data recording: The application offers simple and intuitive tools for recording, storing, and visualizing raw sensor data in real time.
3. Customization: Focuses on delivering a lightweight, customizable sensor configuration system that interacts with a server and database that allows multiple users to register their watches and record multiple sensor data streams simultaneously.
4. Universality: aiming for compatibility with all WearOS devices, beyond those initially provided by dynamically retrieving all registered sensors tailored for each device, ensuring our application remains inclusive and future-proof.

The absence of comprehensive frameworks for recording sensor data presents a notable challenge. Furthermore, with WearOS emerging as a dynamic and rapidly evolving environment, coupled with utilizing Kotlin as the primary programming language, developers often encounter issues such as deprecation and compatibility concerns. *Peppermint* bridges this gap by striving to create an up-to-date application tailored specifically for raw sensor data extraction.

Approach

Research Phase

Sensor Types

There are three main categories of sensors supported by Android: motion sensors, environment sensors, and position sensors [8]. Additionally, there are two types of sensors: hardware-based sensors, which are the physical components built into the wearable and derive the data directly measuring specific environment properties, and software-based sensors, which mimic hardware-based sensors and derive data from one or more hardware-based sensors.

Motion Sensor

Motion sensors allow you to monitor the device movements provided by the user, such as tilt, shake, rotation or swing. In the first case, the device's motion is monitored relative to the frame of reference of the device. In the second case, the motion is relative to the world's frame of reference. Additionally, motion sensors are not used with other sensors, such as the geomagnetic field sensor, to determine the device's position relative to the world's frame of reference. Lastly, a multidimensional array of sensor values is returned for each SensorEvent. The four main sensor properties of the motion sensor include rotation vector, gravity, accelerometer and gyroscope. The rotation vector sensor and gravity sensor allow the detection of gestures and monitor angular change and relative orientation change. The accelerometer sensor is used to measure the acceleration of the device, which includes the force of gravity. If the user wants to measure the real acceleration of the device, excluding the force of gravity, a high-pass filter can be applied. A low-pass filter can be applied to get the force of gravity. The gyroscope sensor measures the rotation rate around the device’s x-axis, y-axis and z-axis in rad/s. Raw rotational data is displayed by the standard gyroscope, which excludes any filtration or correction for noise and drift (bias). The noise and bias are determined by monitoring other sensors such as gravity or accelerometer.

Environment Sensors

Compared to motion and position sensors, the environment sensors return a single sensor value and do not require any data filtering/processing. There are four hardware-based sensors provided by the Android framework to monitor environmental properties. These include relative ambient humidity, illuminance, ambient pressure and ambient temperature. The light, pressure and temperature sensors are acquired from light, pressure and temperature and usually require no calibration, filtering or modification. The humidity sensor acquires raw sensor data by using the same method as light, pressure and temperature. Although the device has a humidity sensor and a temperature sensor, these can be used to calculate the dew point and absolute humidity.

Position Sensors

Position sensors are used to determine the device's physical position in the world's frame of reference. There are three hardware-based sensors provided by the Android Framework to determine the position of the device: geomagnetic field, accelerometer and proximity. The Game Rotation Vector sensor does not use a geomagnetic field, making the relative rotations more accurate as magnetic field changes do not impact them. The geomagnetic rotation vector uses a magnetometer instead of the gyroscope, making the accuracy recorded lower than the normal rotation vector sensor. The geomagnetic field sensor allows monitoring of the changes in Earth's magnetic field and provides raw field strength data for each of the three coordinate axes. The uncalibrated magnetometer sensor is similar to the geomagnetic field sensor, except there is no hard iron calibration applied to the magnetic field. The proximity sensor determines how far an object is from a device.

Android Environment/Implementation

The application was developed on Android Studio [6] and was implemented in Kotlin, Java and XML. When researching, it was discovered that Kotlin is the best language to develop an application for Google Pixel Watches. This programming language is concise and expressive as it reduces common code errors and can easily be integrated into existing applications. As Kotlin is interoperable with Java, it was easier to understand and develop the application. The XML files in the code are the UI of the watch application and show the connectivity between the pages. Also, when creating an application, it is crucial to consider all the dependencies which allow one class to reference another.

Elasticsearch

Elasticsearch is a modern search and analytics engine based on Apache Lucene. It is built on Java and is a NoSQL database which means that the data is stored in an unstructured way and cannot use SQL to query it. This engine can be installed into a Gradle project using Jackson by adding dependencies. The project can then be connected to the Elastic Cloud using an API key and the Elasticsearch endpoint. In this project, all the data recorded by the watches is displayed on Elasticsearch for the user to conduct future research.

Server

The server is implemented using Flask, which allows developers to create an application using Python and HTML/CSS. A virtual environment was used to manage the dependencies of the project and an environment for the server to run on. Once a virtual environment was set up, a flask was installed and was used to run the server. For the server to run, the IP address displayed in the terminal had to match the *baseURL* in the *MainViewModel* file. The server has five different pages: the homepage, login page, registered watches page and the configuration page. The UI for all these pages was created in HTML/CSS.

Scheduled Meetings

As a team, we met twice a week for two hours and discussed the implementation of the application. Our Tuesday meeting primarily focused on what we have developed in the last week, any issues that must be fixed before the meeting with the supervisors, any questions that need to be asked to the supervisors and last-minute preparations before the meeting with the supervisors. Our Thursday meeting focused on the suggestions and points which were made by the supervisors on the previous day. This is when we would jot down our understandings, look at the documentation on how the new elements can be implemented and potentially create diagrams on the whiteboard to ensure that we are both on the same page. Near the end of the meeting, we would divide the work for the week so each person has something to work on. Depending on the week, we would also code parts of the project during our meeting.

Our Wednesday meetings with the professors focused on showing them what we have completed so far. This was done through giving demos, discussing ideas, asking questions and presenting any incurred issues. This meeting ensured that what was being developed followed the project's requirements.

Github Sprints, Goals

On GitHub, we created a roadmap which gave a visual representation of what must be done in each project sprint. The first sprint focused on setting up the environment, watches and Elasticsearch, researching the sensors that can be implemented and data visibility and creating mock-ups for the UI. In the second sprint, we began implementing the sensors on the watch, creating the application's UI and implementing connectivity of the buttons and pages. The third sprint focused on creating a server where users could choose the watch they wanted to configure and enter their Elasticsearch credentials. Additionally, in this sprint, we implemented the connectivity between the watch, server and Elasticsearch to make the application fully functional. The last week of this sprint focused on giving demos, working on final deliverables and preparing for the Creative Computing Day.

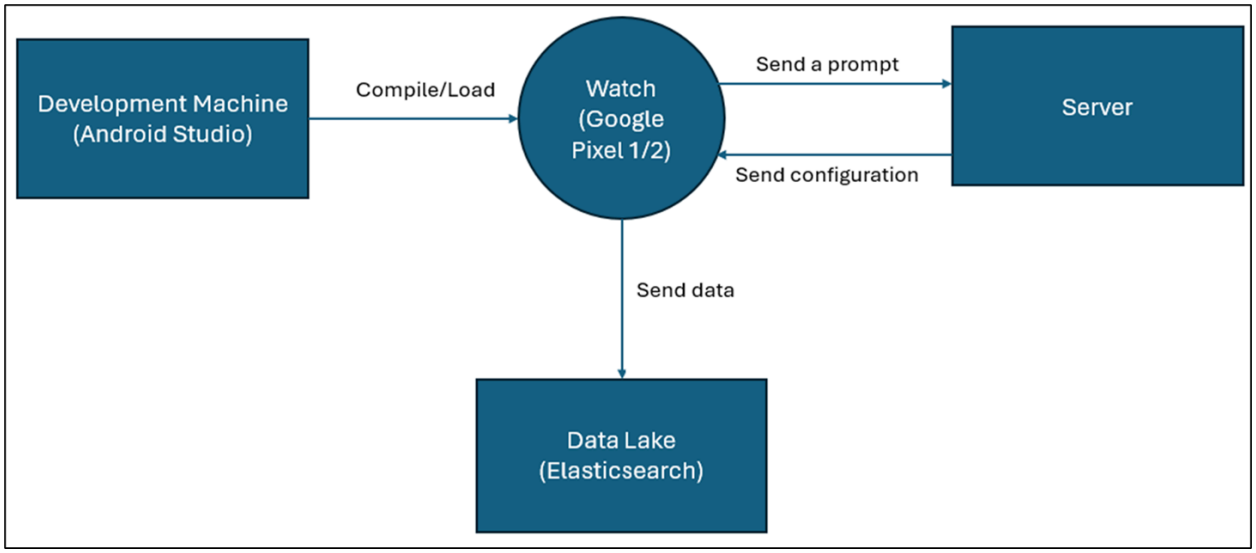

Figure 1.0 - General Flow of the Application

Figure 1.0 showcases the application's general flow, highlighting the primary components of the implementation. The flow begins with the *Development Machine* (in this case Android Studio) where the application is developed. Once this component complies and loads, the application on the *Watch* (Google Pixel 1/2) begins. The *Watch* then sends a prompt to the *Server* and the *Server* sends the selected configuration to the watch. Finally, all the data the watch records is sent to the *Data Lake* (Elasticsearch).

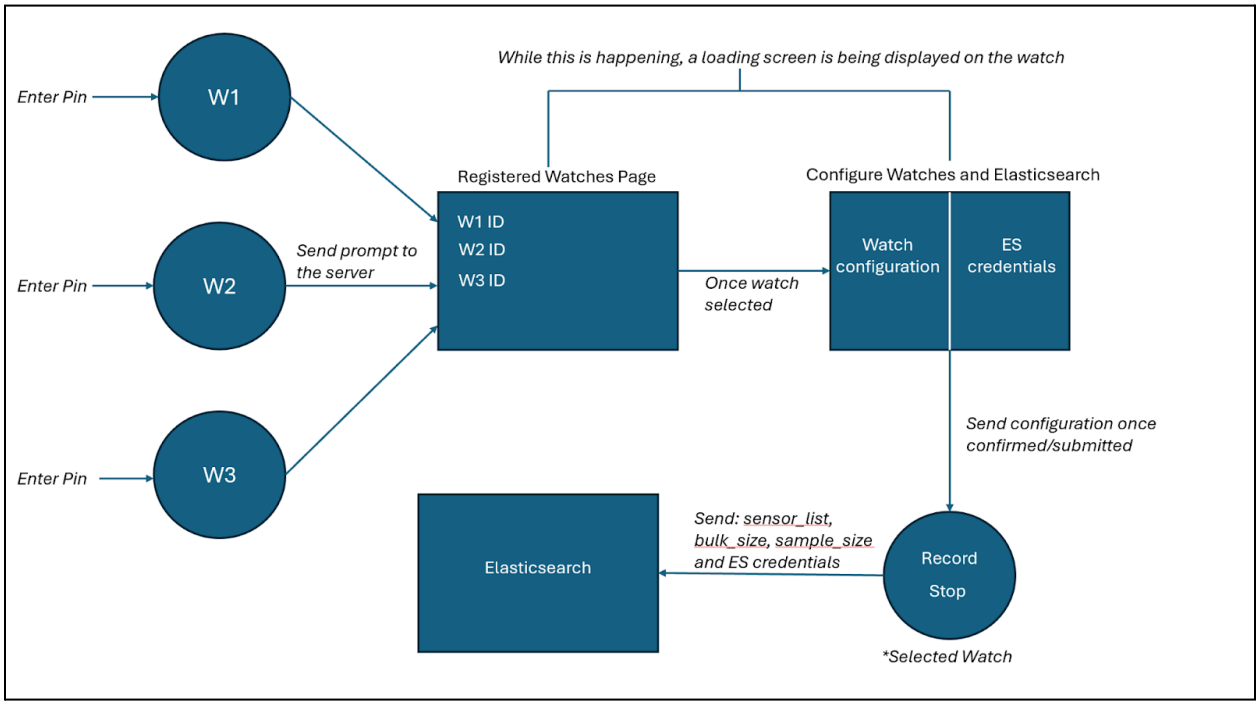

Figure 2.0 - Detailed flow of the Applications

Figure 2.0 showcases a detailed flow of the application. First, the user enters a PIN on the watch, which sends a prompt to the server indicating that the watch is now connected to the server. The user can then select the connected watch and configure it on the next page. While the user enters this information on the server, a loading screen is displayed on the watch. Once the user clicks submit on the server, the user can begin recording on the watch. The data recorded is then sent to Elasticsearch, where it is displayed.

Results

App Development

As a team, we created an application for Google Pixel 1 and Google Pixel 2 watches. The application allows users to exploit the watch's sensor types and extract their raw sensor data. This application was implemented in Android Studio using Kotlin, Java and XML.

USER INTERFACE:

The *MainActivity.kt* class focuses on starting up the screen for the application. It allows the user to enter a PIN verification, which allows the device to connect to the server and exploit all the sensor types. Once the user enters the correct credentials, it is directed to the *LoadingActivity.kt* file or the *SensorActivity.kt* screen. The user will continue to see these screens until the server is shut down or the user times out (after 3 min). If the user enters the incorrect credentials, the screen will display an error message stating that the PIN is incorrect.

The *LoadingActivity.kt* class focuses on displaying the loading animation screen while the user is waiting for the server to send the prompt to the watch or the server is down but the WearOS is still attempting to connect to the server.

The *SensorActivity.kt* class displays the screen that the user is directed to once it has connected to the server correctly. Four primary functions that are being achieved here:

1. Constant Loop: Ask the user which sensors (including other configurations) they want to record the data for from the server
2. Constant Loop: Ask for confirmation on the selected sensors and configuration to record from the server
3. Constant Loop: Ask for Elasticsearch client credentials that the data should be sent to
4. Record/stop the data recording session. The real-time data is then sent to Elasticsearch

The layout of the pages is embedded in their respective XML files, which include *activiyt_main.xml*, *activity_sensor.xml* and *loading_activity.xml*.

SENSOR

The *SensorDataManager* class focuses on managing the sensor data and its business logic for the WearOS device. This class can achieve three main functionalities.

1. It parses the response body from the server to retrieve configurations selected by the user. This includes the sensor list (what to record), sample rate, bulk size and Elasticsearch credentials (URL, username, password and index)
2. Sensor listeners are registered/unregistered according to the sensor list from the response body
3. Performs the indexing operations to Elasticsearch (via ESIndexOperations)

The *SensorData.java* class initializes all the data objects being used in the application.

MODEL

The *MainViewModel* belongs to the pre-existing ViewModel class. The purpose of the ViewModel class is to manage UI-related data in a lifecycle-conscious way and is responsible for preparing and managing data for an activity or fragment.

Dataflow Server Connection

SERVER IMPLEMENTATION

For the application to function, it is connected to a local Flask server. The server consists of four pages: homepage, login page, registered watches page and configuration page. All these pages were created using HTML, CSS and JavaScript (for functionality). For the watch to be connected to the server, the user has to enter their login credentials on the login page and enter the four-digit PIN on the watch. Once the watch is connected, the user can select the watch on the registered watches and configure it on the next page by selecting the sensor types, entering the bulk size, sample size and Elasticsearch credentials. This will ensure that all the data recorded through the watch goes to the defined Elasticsearch credentials.

RUNNING THE SERVER

To run the server, the user must first install and activate the virtual environment in the project folder. Once activated, the user can then run the flask command. It is important to note that the *baseServerURL* in the *MainViewModel* should be identical to the link displayed in the terminal when the user runs the server.

Elasticsearch Results

Elasticsearch is a search and analytics engine which is used for log analytics, full-text search, security intelligence, business analytics and operational intelligence use cases. This engine stores a document, which is then indexed and becomes fully searchable in near real-time (within 1 second) [2]. The data collected by the watch for the specific sensors is displayed on Elasticsearch. To access the data, the user must enter their Elasticsearch credentials including the *Elasticsearch URL, Username, Password and Index* on the *Configuration Page* of the server. Once the user opens the Elasticsearch URL, the document on that page displays the information for each watch connected to the server. The information includes *dataValues[x,y,z]* (the data values depend on the sensors selected by the user - sensor API), *name* of the sensor, *timestamp, type* of sensor*, watchID, _id, _index, score* and *session_ID*. Additionally, to view the results in real-time (while the data is being recorded), the user can run the *es_op2.py* file, and to see a visual representation of the results, the user can run the *es_op.py* file.

*Specific Sensor*s

WearOS (formerly known as Android Wear) is a version of Google's Android operating system designated for smartwatches and other wearables. It can be paired with mobile phones running Android version 6.0 "Marshmallow" or later. Our scope focuses on the Google Pixel Watches, which are not supported by iOS version 10.0 or newer with Google's pairing application. Below are the specifications for both watches that were used during the experiments [4]:

Google Pixel Watch 1

Category	**Specifications**
Connectivity	● 4G LTE and UMTS ● Bluetooth® 5.0 ● Wi-Fi 802.11 b/g/n 2.4 GHz ● NFC
Compatibility	● Most Android 8.0 or newer
Power	● 294 mAh ● Built-in rechargeable lithium-ion battery
Chip	● Exynos 9110 SoC ● Cortex M33 co-processor
OS	● Wear OS 3.5
Storage and Memory	● 32 GB eMMC FLASH

	● 2 GB SDRAM
Sensors	● Compass ● Altimeter ● Blood oxygen sensor ● Multipurpose electrical sensor ● Optical heart rate sensor ● Accelerometer ● Gyroscope ● Ambient light sensor

Google Pixel Watch 2

Category	**Specifications**
Connectivity	● 4G LTE and UMTS ● Bluetooth® 5.0 ● Wi-Fi 802.11 b/g/n 2.4 GHz ● NFC
Compatibility	● Most Android 9.0 or newer Requires Google Account and Google Pixel Watch app
Power	● 306 mAh ● Built-in rechargeable lithium-ion battery
Chip	● Qualcomm SW5100 ● Cortex M33 co-processor
OS	● Wear OS 4.0
Storage and Memory	● 32 GB eMMC FLASH ● 2 GB SDRAM
Sensors	● Compass ● Altimeter ● Red and infrared sensors for oxygen saturation (SpO2) ● Multipurpose electrical sensors compatible with ECG app ● Multipath optical heart rate sensor ● 3-axis accelerometer ● Gyroscope ● Ambient light sensor ● Electrical sensor to measure skin conductance (cEDA) for body response tracking ● Skin temperature sensor ● Barometer ● Magnetometer

Battery Consumption

In this experiment, our focus was to assess the battery performance of the Google Pixel Watch (1 and 2) within the context of utilizing the raw sensor data extraction. Given the integral role of battery life in wearable technology, understanding its implications for applications such as raw sensor data extraction is crucial. These experiments were designed to provide insights into how the battery life of the Google Pixel Watch (1 and 2) is impacted by the operation of a raw sensor data extraction application, thereby shedding light on its practical implications and importance for users relying on functionalities. To comprehensively evaluate battery performance, we conducted a series of three experiments for each watch model – monitoring its performances using Android's *Energy Profiler* within the Android IDE [5]. Firstly, we recorded data from a single sensor to gauge the battery drain under minimal load. Secondly, we expanded the scope by recording data from multiple sensors simultaneously, simulating a more intensive usage scenario. Finally, we observed battery behaviour when the application was inactive (not recording), serving as a baseline for comparison. All experiments are run for a 30-minute duration. Additionally, by maintaining the screen brightness at its default level, we aim to capture a realistic representation of how the application impacts overall battery performance under standard operating conditions. It is worth noting that the overhead may differ between local machine mirroring and standalone production build environments.

Google Pixel (Gen 1)

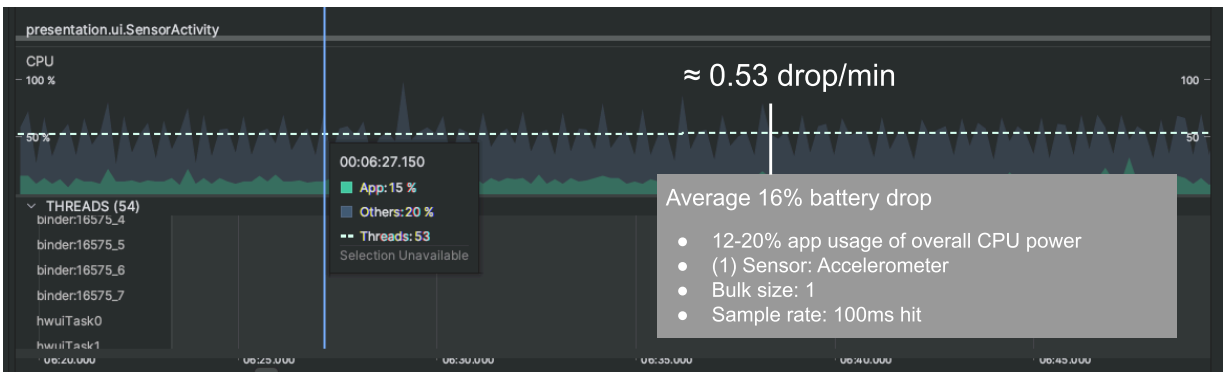

Figure 3.0 - Google Pixel (Gen 1) Experiment #1: One sensor recording

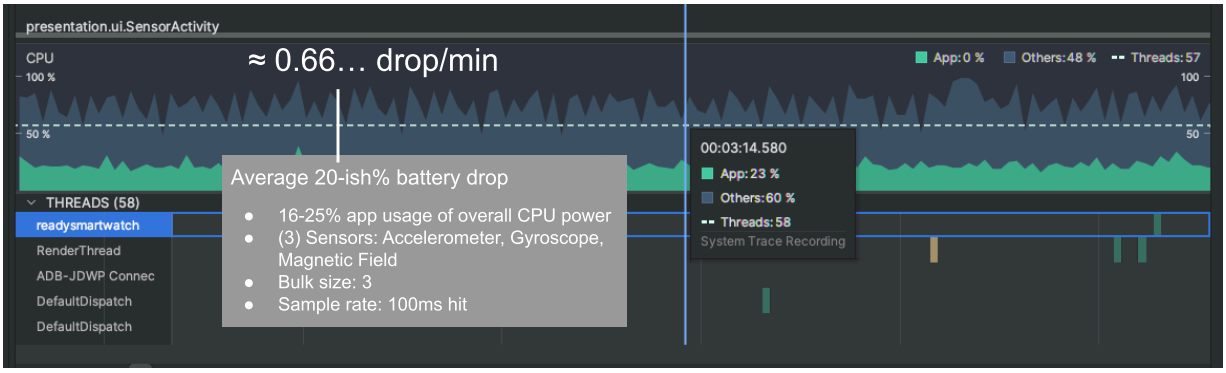

Figure 3.1 - Google Pixel (Gen 1) Experiment #2: Three sensors recording

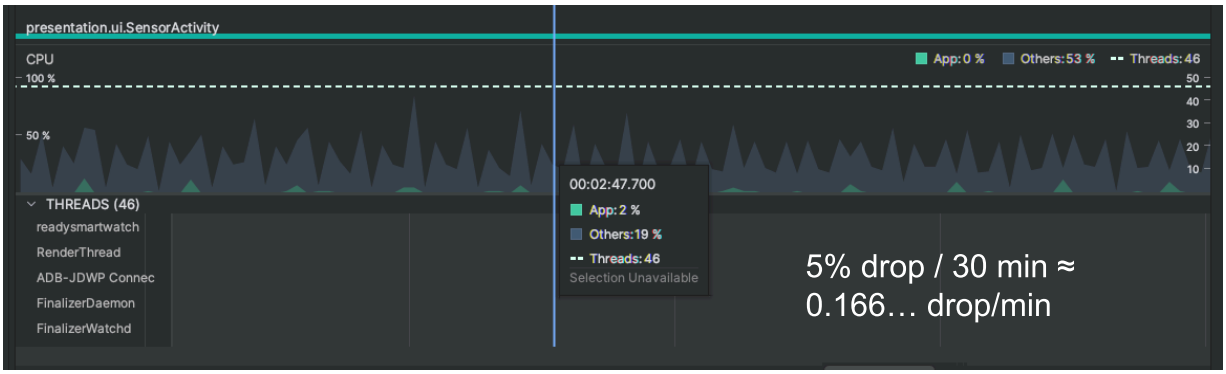

Figure 3.2 - Google Pixel (Gen 1) Experiment #3: Idle Sensor Activity

In the third experiment, the user is idle on the sensor activity screen. Most of the time, there is almost 0% app usage of overall CPU power. The sporadic small spikes in CPU usage observed during idle screen periods on the WearOS app may be attributed to background processes since the application continuously fetches sensor configuration changes. Furthermore, the application is constantly pinging to the server endpoint every 2 seconds seeking if there are any changes to what to record. Upon initiation of the recording process in both experiments, significant CPU power and battery consumption were observed in its first few seconds. This behaviour arises due to the execution of background processes, primarily involving establishing connections to the Elasticsearch servers and the initial uploading of the sensor data. This extensive power usage slowly declines to a consistent pattern afterwards. It is expected that capturing multiple sensor data streams versus the single stream uses more power; with a difference of approximately 5% CPU power with 4% battery drainage.

Google Pixel (Gen 2)

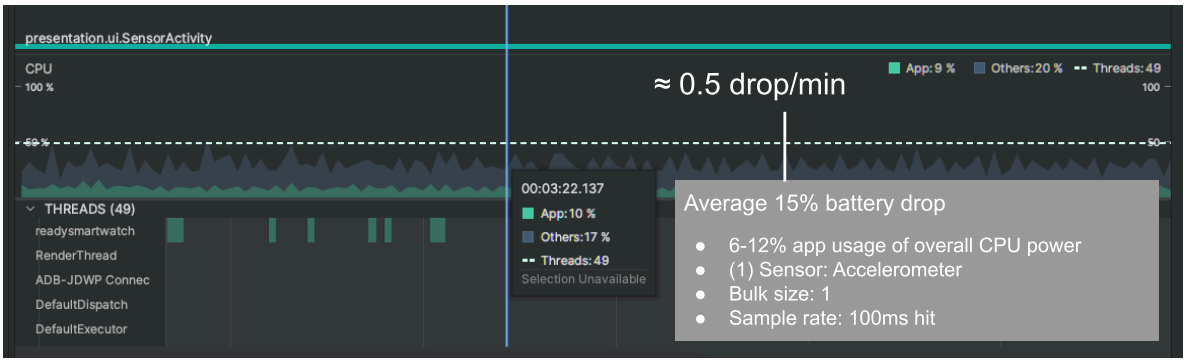

Figure 4.0 - Google Pixel (Gen 2) Experiment #1: One sensor recording

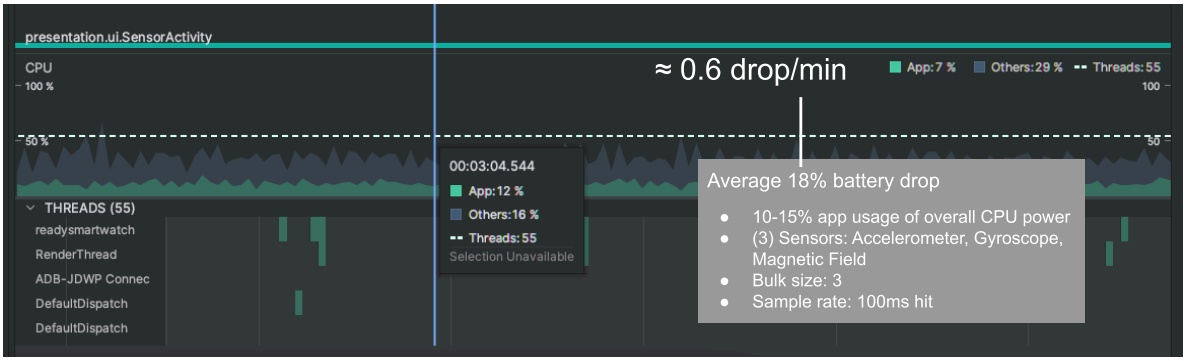

Figure 4.1 - Google Pixel (Gen 2) Experiment #2: Three sensors recording

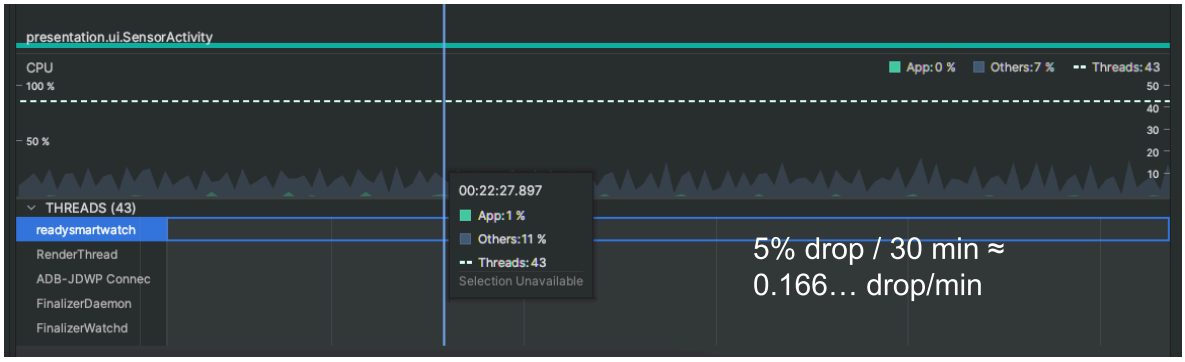

Figure 4.2 - Google Pixel (Gen 2) Experiment #3: Idle Sensor Activity

The Google Pixel Gen 2 has a slightly better battery efficiency than the Gen 1. This is due to its battery upgrade from the previous iteration. Additionally, its sensor types use less battery power (i.e. the magnetic field sensor on the Pixel 2 utilizes the LIS2MDL, which is an ultra-low-power, high-performance 3-axis digital magnetic sensor departing from its predecessor).

Latency

In the sensor activity screen for the WearOS app, the UI timer is crucial in providing users with a general estimation of elapsed time during data retrieval processes. However, it is important to note that this timer does not precisely align with the actual duration of sensor data retrieval. In previous experiments, a sampling rate of 100,000 (equivalent to a hit per 100ms) was used. While the time logs on the emulator were fully accurate, this rate does not precisely reflect the sampling interval on the exact millisecond for both watches. This behaviour likely occurs due to other background apps running on the real watches while the emulator focuses on the tested application. Nevertheless, it is noteworthy that despite potential discrepancies in sampling rates, the watches are logging data at a higher frequency, surpassing the 100ms hit threshold. This higher sampling rate frequency is advantageous as it captures sensor variations at more frequent intervals, thereby enhancing the granularity and accuracy of the collected data.

Figure 5.0 - Raw Sensor Data Logs on Elasticsearch – Running 100ms Sampling Rate

```
Timestamp: 2024-02-06 18:58:22.256 Data Values: [-0.41920874, 4.3861213, 8.750683]
Timestamp: 2024-02-06 18:58:21.610 Data Values: [0.89590895, 6.376764, 18.567354]
Timestamp: 2024-02-06 18:58:21.693 Data Values: [-1.27679, 5.127522, 13.926114]
Timestamp: 2024-02-06 18:58:21.774 Data Values: [-0.33416927, 4.6639967, 10.408354]
Timestamp: 2024-02-06 18:58:21.855 Data Values: [-0.22517498, 4.6735787, 9.590299]
Timestamp: 2024-02-06 18:58:21.934 Data Values: [-0.25511846, 4.59333, 8.830932]
Timestamp: 2024-02-06 18:58:22.014 Data Values: [-0.3233896, 4.4400196, 8.500356]
Timestamp: 2024-02-06 18:58:22.095 Data Values: [-0.24673429, 4.792155, 8.881237]
Timestamp: 2024-02-06 18:58:22.178 Data Values: [-0.45514092, 4.4891267, 8.416513]
Timestamp: 2024-02-06 18:58:22.256 Data Values: [-0.41920874, 4.3861213, 8.750683]
Timestamp: 2024-02-06 18:58:22.661 Data Values: [-0.2682936, 4.539432, 8.506344]
Timestamp: 2024-02-06 18:58:22.742 Data Values: [-0.07665531, 4.7526293, 8.517124]
Timestamp: 2024-02-06 18:58:22.824 Data Values: [-0.0071864356, 4.611296, 8.68361]
Timestamp: 2024-02-06 18:58:22.903 Data Values: [-0.22158177, 4.515477, 8.579407]
Timestamp: 2024-02-06 18:58:22.984 Data Values: [-0.08983044, 4.6412396, 8.695587]
Timestamp: 2024-02-06 18:58:23.064 Data Values: [-0.046711832, 4.6412396, 8.647677]
Timestamp: 2024-02-06 18:58:23.144 Data Values: [-0.22637272, 4.534641, 8.615338]
Timestamp: 2024-02-06 18:58:23.227 Data Values: [-0.16409028, 4.588539, 8.608152]
Timestamp: 2024-02-06 18:58:23.308 Data Values: [-0.025152525, 4.6412396, 8.621327]
Timestamp: 2024-02-06 18:58:23.631 Data Values: [-0.25152525, 4.508291, 8.591384]
Timestamp: 2024-02-06 18:58:23.711 Data Values: [-0.21319759, 4.612494, 8.610548]
Timestamp: 2024-02-06 18:58:23.791 Data Values: [-0.2503275, 4.4783473, 8.689598]
Timestamp: 2024-02-06 18:58:23.872 Data Values: [-0.31500542, 4.6196804, 8.651271]
Timestamp: 2024-02-06 18:58:23.953 Data Values: [-0.32458735, 4.575364, 8.597372]
```

Figure 6.0 - Python Monitoring Script

For data log monitoring, we implemented a Python script designed to fetch the most recent documents efficiently. The script is configured to retrieve the latest 9-10 data logs at intervals of 500ms. This approach has demonstrated remarkable effectiveness, evidenced by the near-instantaneous retrieval of the latest logging

details. Notably, actions such as moving the watch up and down yield almost instantaneous results, confirming the real-time responsiveness of our monitoring system. This streamlined process underscores that the data is logged on Elasticsearch in a pseudo-real-time manner.

Additionally, we've implemented a streamlined data visualization feature leveraging Matplotlib, which elegantly renders sensor data alongside their respective timestamps. Although there is a nominal 2-3 second delay in fetching and updating the graph, it's important to note that within the broader project framework, the intention is to seamlessly integrate with Kibana – a powerful tool synergizing with Elasticsearch for visually stunning data representation. This transition ensures a seamless and aesthetically pleasing display of the data, enhancing overall user experience and analytical capabilities.

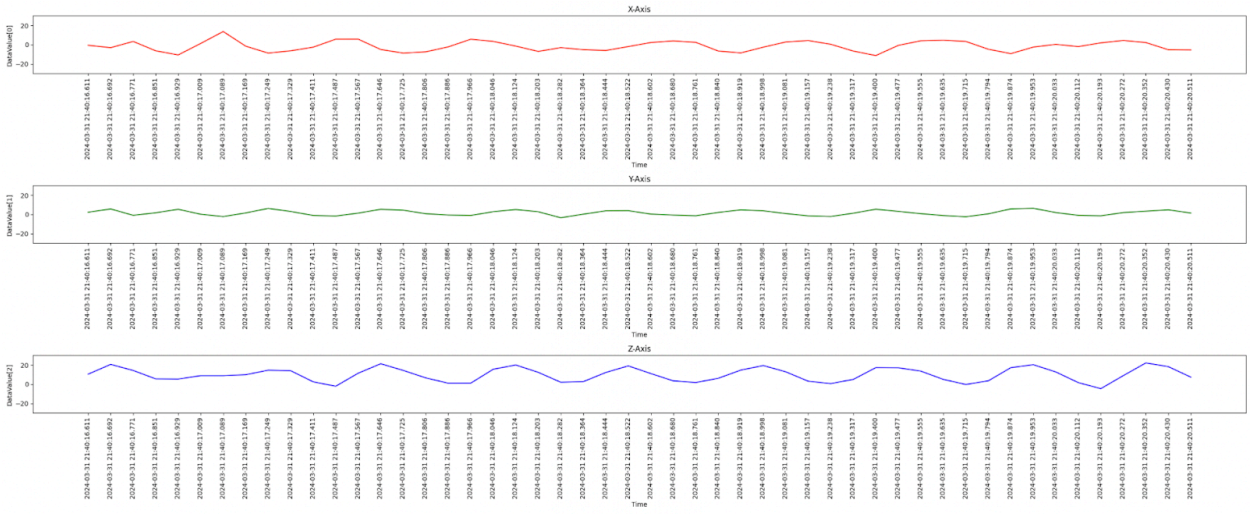

Figure 7.0 - Data Visualization of the Accelerometer Sensor Using Matplotlib

Problems / Challenges

1. **Interoperability:** Operating between multiple languages, such as Kotlin, Java, and Python, presented a series of challenges and complexities for the development team working within the WearOS app development environment. One of the primary difficulties encountered is the need for seamless integration and interoperability between these languages. Kotlin's interoperability, such that it can seamlessly interact with existing Java libraries, frameworks, and code is a crucial aspect that facilitates the transition between these languages within the same codebase environment. The decision to leverage Kotlin and Java in our application, rather than just using the widely supported use of Java as the standalone language, stems from various factors. Kotlin, despite being a relatively newer language than Java, offers several advantages, such as concise

and digestible syntax, null safety, and modern features. Additionally, Kotlin is expected to be the leading language adoption in the software development community, as it is fully supported by Google for Android app development. However, there are instances where the use of Java becomes necessary, particularly when interfacing with certain third-party libraries and frameworks that are only available in Java. One such example is the need to create Elasticsearch clients – from the Elasticsearch API only supported by a few languages [3]. Parsing the data into JSON format and retrieving this data from Elasticsearch using our Python scripts also presented some challenges.

2. **Deprecation Issues:** Navigating deprecation issues posed significant challenges. The use of Java in the context of Elasticsearch has undergone many updates that have resulted in mass deprecation, leading to diminishing support and documentation availability. Overcoming this obstacle required considerable effort, as we diligently exploited resources beyond official websites – inspecting other developers' code and endeavored to adapt it to our specific requirements. This challenge is further amplified by the rapidly evolving nature of Kotlin and the use of third-party applications such as JWS tokenization, OkHTTP, and JSON serialization, resulting in a scarcity of up-to-date resources. Additionally, wearables presented a unique environment distinction from conventional mobile phone devices. Android API aims to maintain universality between its code and devices; however, the limited screen space and power from the wearables necessitated specialized development practices. Consequently, documentation published beyond a year may become outdated, posing difficulties in maintaining and updating WearOS applications effectively.

3. **Directions:** In embarking on this project, we found ourselves starting from scratch, disadvantaged by any previous projects that could provide foundational insights. As a result, we undertook the task of laying the groundwork ourselves. Initially, our vision involved putting all functionalities directly onto the wearable device. This encompassed features like the login page and the configuration of sensor settings, including bulk size, sample rate, and data recording preferences, all managed solely on the watch interface. However, upon careful consideration, we opted for a different approach – concluding that establishing a server to facilitate these processes would offer greater advantages. This server-centric architecture proved lighter on the wearable and inherently more modular, offering flexibility for future iterations and enhancements. Nonetheless, this decision introduced a new set of challenges. Now facing the task of establishing communication channels between the device and a standalone server. The server, in turn, would

dictate the recording parameters to the device, with subsequent data transmission back to the watch and recordings sent directly to a designated database server (Elasticsearch). In doing so, the project's workflow became considerably more intricate, demanding extensive attention and robust implementation strategies to navigate the complexities introduced by this architectural shift.

Future Work

Foreground Services

Initially, the concept under consideration was implementing a sleep mode for WearOS devices, wherein the screen would remain off or exit out of the app and have the background operation running to save significant battery life. However, this approach necessitated a restructuring of our codebase to achieve the desired functionality. Looking ahead, for further enhancement, it is highly suggested to leverage Android's Foreground Services. This API enables the execution of long-running tasks without the requirement of keeping the UI displayed at all times. Furthermore, this service allows developers to create functionalities that run in the foreground, meaning they have a higher priority than background services that are less likely to be terminated by the system. The primary objective behind exploring such implementation was to significantly extend the battery life without compromising user experience. By allowing the screen to remain off or exit the app while background operations continue to run, we aim to conserve power consumption during idle periods. Extending beyond battery life optimization, this purpose suggests a realistic standard for UX. For instance, continuous authentication processes necessitate background operation without explicit app display. By enabling the execution of long-running tasks without the need for constant UI presence, this API aligns with the evolving demands of wearable technology, facilitating prioritized execution of critical functionalities while ensuring seamless multitasking and uninterrupted user interactions.

Security

Flask is a lightweight and flexible framework for Python; however, out of the scope of this project, the future is to aim for enhanced security measures. While its framework provides a solid and straightforward foundation for web development, prioritizing its security is crucial to ensure the integrity and confidentiality of data processed by the application. Firstly, Flask's session management mechanism exhibits notable weaknesses, which can lead to session hijacking or fixation. Furthermore, our application reveals insecure session cookie management and insufficient session expiration policies. This can increase the likelihood of unauthorized access to sensitive user sessions. For the future, strengthening session management entails the adoption of secure session cookie configurations, rigorous expiration protocols, and encryption measures, particularly over HTTPS, to safeguard data during transmission.

Moreover, the authentication mechanism deployed within our Flask application lacks robustness. For instance, our 4-digit PIN verification to make a connection to the server and session management is susceptible to brute-force attacks. It is highly suggested to incorporate comprehensive authentication protocols, such as OAuth 2.0 or JSON Web Tokens (JWT) [1], to ensure multifactor authentication, password hashing, and account lockout mechanisms to fortify user authentication processes, ultimately deterring unauthorized access attempts effectively. Additionally, we have set up a temporary configuration file of the Flask application that presents a significant breach in security protocols. Critical information, including Elasticsearch database credentials, server account information, and PIN verification, are inadequately protected. It is highly advised to use environment variables, separate user databases that require authentication, or token-based authentication instead of hardcoding sensitive information and getting the server to extract the file.

App Deployment

Currently, to run the application, the user has to run the project in Android Studio. This can be quite cumbersome for the user and may propose limitations for users who don't have the application installed. To improve this, the application can be sideloaded so the user doesn't have to run the program on Android Studio whenever they want to record data. To sideload the application, the user must download the installation file, which is an *.APK* file extension (Android Package). An APK file can be created in Android Studio by going to the "Build" menu and selecting "Build Bundle(s)/APK(s)" and then "Build APK(s)". Next, wait for the Gradle to build. Once the build process has been completed, a message indicating the location. APKs can be installed from websites such as APK Mirror and APKPure. Once downloaded, sideloading can officially begin. The APK file or the app bundle can be transferred from the computer to the watch by connecting it through a USB cable. ADB can be used to sideload app bundles (ensure that ADB is installed). The app bundle can be installed using it and the contents can be extracted to a folder. The following command can be used to install the app bundle [7], [9]:

adb install-multiple one.apk **two**.apk **three**.apk

The extracted APK files should be in the platform-tools or ADB folder for easy access. The command will allow the application to be installed on the watch. **To set up a build for our Android app development, refer to our GitHub and its requirements**.

References

[1] “Authenticate to Oauth2 Services: Android Developers.” *Android Developers*, developer.android.com/training/id-auth/authenticate. Accessed 7 Apr. 2024.

[2] *Data in: Documents and indices: Elasticsearch Guide [8.13]*. Elastic. (n.d.). https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html#:~:text=Elasticsearch%20uses%20a%20data%20structure,documents%20each%20word%20occurs%20in.

[3] “Elasticsearch Java API Client.” *Elastic*, www.elastic.co/guide/en/elasticsearch/client/java-api-client/current/index.html. 7 Apr. 2024.

[4] “Google Pixel Watch Technical & Device Specifications.” *Google Pixel Watch Help*, Google, support.google.com/googlepixelwatch/answer/12651869?hl=en#zippy=%2Cgoogle-pixel-watch-specifications. 7 Apr. 2024.

[5] “Inspect Energy Use with Energy Profiler: Android Studio: Android Developers.” *Android Developers*, developer.android.com/studio/profile/energy-profiler. Accessed 7 Apr. 2024.

[6] “Install Android Studio: Android Developers.” *Android Developers*, developer.android.com/studio/install. Accessed 7 Apr. 2024.

[7] Msowski. (2023, September 4). *Release your App’s APK file and deploy it on your device*. Medium. https://medium.com/@msowski/release-your-apps-apk-file-and-deploy-it-on-your-device-66a5a2177ac8

[8] “Sensors Overview: Sensors and Location: Android Developers.” *Android Developers*, developer.android.com/develop/sensors-and-location/sensors/sensors_overview. 7 Apr. 2024.

[9] Siddiqui, A. (2022, October 28). *How to sideload and install apps on Android as apks or App Bundles*. XDA Developers. https://www.xda-developers.com/how-to-sideload-install-android-app-apk/

[10] S. Mare, A. M. Markham, C. Cornelius, R. Peterson, and D. Kotz, ‘‘ZEBRA: Zero-effort bilateral recurring authentication,’’ in *Proc. IEEE Symposium on Security and Privacy*, May 2014.