HifoCap: An Android App for Wearable Health Devices

Yoonsik Cheon, Rodrigo Romero and Javier Garcia

TR #16-63
September 2016; revised January 2017

Keywords: electroencephalogram (EEG), health-care app, space and time efficiency, wearable devices, Android, HifoCap.

1998 CR Categories: D.2.2 [*Software Engineering*] Design Tools and Techniques—object-oriented design methods; D.2.3 [*Software Engineering*] Coding Tools and Techniques—object-oriented programming; D.3.3 [*Software Engineering*] Language Constructs and Features—classes and objects, dynamic memory management, input/output; D.4 [*Operating Systems*] Storage Management—garbage collection; D. J.3 [*Life and Medical Sciences*] Health.

To appear in *the Eighth International Conference on the Applications Digital Information and Web Technologies (ICADIWT 2017), Juarez City, Mexico, March 29-31, 2017*.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

HifoCap: An Android App for Wearable Health Devices

Yoonsik CHEON[a,1], Rodrigo ROMERO[b] and Javier GARCIA[a]
[a] *Dept. of Computer Science, The University of Texas at El Paso (UTEP), El Paso, TX*
[b] *Dept. of Electrical and Computer Engineering, UTEP, El Paso, TX*

Abstract. Android is becoming a platform for mobile health-care devices and apps. However, there are many challenges in developing soft real-time, health-care apps for non-dedicated mobile devices like smartphones and tablets. In this paper, we share our experiences in developing the HifoCap app, a mobile app for receiving electroencephalogram (EEG) wave samples from a wearable device, visualizing the received EEG samples, and transmitting them to a cloud storage server. The app is network and data-intensive. We describe the challenges we faced while developing the HifoCap app—e.g., ensuring the soft real-time requirement in the presence of uncertainty on the Android platform—along with our solutions to them. We measure both the time and space efficiency of our app and evaluate the effectiveness of our solutions quantitatively. We believe our solutions to be applicable to other soft real-time apps targeted for non-dedicated Android devices.

Keywords. electroencephalogram (EEG), health-care app, space and time efficiency, wearable devices, Android, HifoCap

1. Introduction

Advances in mobile devices as well as their software platforms—e.g., faster processors, bigger storage, larger screen, smaller batteries, various sensors, and open-source operating systems—have paved the way for the development of a flood of medical mobile devices and apps [1]. One can measure one's own blood pressure, use a portable ultrasound, or spit on a diagnostic kit for sexually transmitted diseases and shortly have the results on one's smartphone screen. This became possible by connecting to one's smartphone or tablet various types of sensors and other medical accessories, e.g., blood pressure monitors and glucose meters. We also have seen a big surge in the adoption of Android-based medical devices and apps because they allow the ability to provide cost effective medical care to patients outside the hospital [1]. However, due to the inherent nature of the Android operating system, there are many interesting challenges in using Android as a platform for mobile medical devices and applications. For example, Android cannot be qualified to be used in real-time environments [2], and thus one unique challenge is to meet the soft real-time requirement of some of the medical apps by making the performance of the apps more efficient and predictable.

An electroencephalogram (EEG) is a medical test, typically performed in a doctor's office or at a hospital, to detect abnormalities related to electrical activities of the brain.

[1] Yoonsik Cheon, Dept. of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX 79968, U.S.A.; Email: ycheon@utep.edu.

With the electrodes placed along the scalp, the test measures voltage fluctuations resulting from ionic current within the neurons of the brain [3]. Normal electrical activity in the brain makes a recognizable pattern, and thus one can look for abnormal patterns that indicate seizures and other problems such as sleep disorders and changes in behavior (see Figure 1).

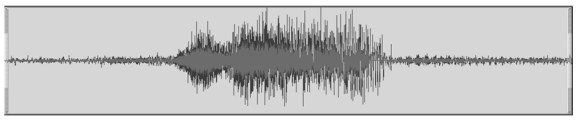

Figure 1. EEG showing epileptic seizure

In this paper, we share our experiences of developing a wearable system called HifoCap for automatic detection of scalp high-frequency oscillations (HFOs), EEG waves with frequencies in a specific range (see Section 2). The HifoCap system consists of a wearable device hidden inside a cap, an app running on a mobile device like a smartphone, and a cloud storage server. The cap senses cortical signals and transmits them to the app. The app analyzes and visualizes the received EEG waves before transmitting them to the cloud server for further off-line analyses. The focus of this paper is on the development of the HifoCap app targeted for non-dedicated Android devices. The app is data-intensive and offers soft real-time performance. It may miss some deadlines without incurring in unacceptable performance degradation. We describe the technical challenges for developing the app and our solutions to them. The main challenge is to assure soft real-time by supporting the required time and space performance. Our discussion focuses on performance optimization and avoids non-technical aspects such as compliance with the U.S. Food and Drug Administration (FDA) and other regulatory standards.

The rest of this paper is structured as follows. In Section 2 below we provide a quick overview of the HifoCap system and its components. In Section 3 we summarize the key requirements of the HifoCap app. We also identify the challenges of developing soft real-time apps on non-dedicated Android devices. In Section 4 and 5, we first propose our solutions to the challenges and then evaluate the effectiveness of our proposed solutions by measuring both the time and space efficiency of our app. In Section 6 we conclude this paper with a concluding remark.

2. The HifoCap System

An EEG test has been traditionally used for research related to neurophysiology and for diagnosing brain disorders such as epilepsy [4]. The standard empirical classification of EEG waves includes delta, theta, alpha, beta, and gamma waves, all below a 100 Hz threshold, and the common practice does not consider waves above 100 Hz to be useful and tends to focus on the beta range of 16 to 31 Hz and ranges below it. However, recently waves with frequencies on the upper gamma range and above, referred to as *high-frequency oscillations* (70 to 500 Hz), ripples (80 to less than 250 Hz), and fast ripples (250 to 600 Hz), have received a great deal of attention from the research community to detect pathological brain activity and to understand cognitive processes as well [5] [6].

While the amplitude of scalp high-frequency oscillations (HFOs) has been found smaller than the amplitude of intracranial HFOs, non-invasive detection of HFOs could increase their use as biomarkers for clinical applications and research. However, due to their low signal levels in relation to other EEG signal components and their potential false positives when using only spectral analysis for detection, HFOs must be detected through a combination of time domain analysis, spectral analysis, and visual inspection. Additionally, HFOs are not continuously present in EEG recordings but appear as short sequences of a few to several high-frequency cycles added over signals in the lower frequency ranges. Thus, detecting HFOs is a time-consuming and potentially error-prone task that could be substantially improved through signal processing with machine learning techniques to remove irrelevant data.

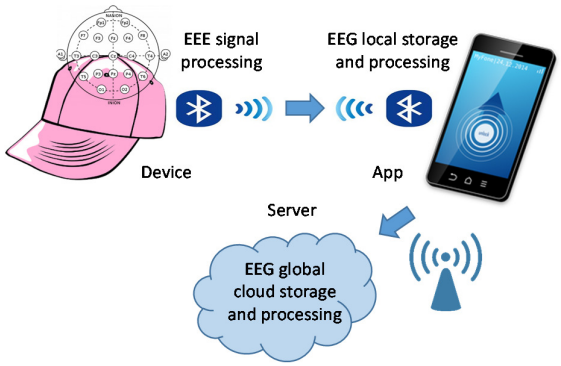

Figure 2. HifoCap system

We have been developing HifoCap, a wearable system for automatic detection of scalp HFOs (see Figure 2). It performs very high dynamic range processing of EEG signals in the time-frequency domain or time-scale (wavelet) domain. A HifoCap device hidden inside a cap senses cortical signals and processes them with an embedded system. The main processing steps include amplification, filtering, and HFO detection. EEG waves containing HFOs are transmitted to a smartphone or tablet wirelessly using a personal area network protocol such as Bluetooth. An app running on the smartphone receives EEG waves for recording, time stamping, logging, plotting, and transmitting wirelessly to a cloud storage server. The cloud server stores EEG waves and supports multiple types of off-line analyses.

Developing the HifoCap system is a research and engineering challenge requiring interdisciplinary work on wearable hardware-software codesign, health data analyses, and human factors and ergonomics. However, we believe that recent advances in digital components, mobile/wearable computing, cloud computing, and big data analysis present an unprecedented opportunity to realize such a system. The benefits of the HifoCap system are immediate and far-reaching, both economically and socially. In addition to cost and convenience for the end user, a very important advantage of our wearable system design is enabling detection of HFOs while the user is away from clinical settings and perhaps just going about his or her usual daily routines. The long-term impact of a system like HifoCap is enormous, considering the fact that epilepsy affects about 50 million people in the world–over three million people in the U.S.–and it limits their ability to lead a happy, productive life. The system will enable epilepsy

research and treatment by providing a means to analyze brain activity with massive field recordings, as opposed to the currently available recordings obtained in a clinical setting while the patient is in an absolute motionless position.

3. Requirements and Challenges

The platform for the HifoCap app is non-dedicated Android devices such as smartphones and tablets with wireless network capabilities such as Bluetooth and Internet access. The key requirements for the app include (see Figure 3).

- To configure and control the HifoCap device for sampling EEG signals.
- To receive EEG signal samples from the HifoCap device and log, time stamp, record, plot, and transmit the samples to the HifoCap server.
- To track daily activities of the user.

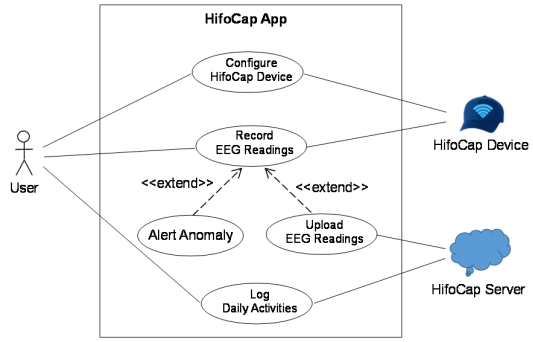

Figure 3. Use case diagram

As medical technology is getting more pervasive, there is a need to interface with a variety of protocols such as Bluetooth and Wi-Fi for data transfer. All communications between the app and other components of the HifoCap system are to be done wirelessly. Bluetooth—which can wirelessly connect devices together—is used for sending control signals to the device and for receiving EEG samples from the device. Wireless Internet technology such as Wi-Fi and a cellular data network is to be used to transmit EEG samples to the storage server.

The HifoCap system is a soft real-time system, though not safety-critical. A soft real-time system may miss some deadlines, but eventually, performance will degrade if too many deadlines are missed. The live EEG plotting requires EEG samples to be streamed from the HifoCap device and then rendered on the display in real-time. Control commands from the app to the device must be processed within a deterministic deadline. In addition, the app must be able to run continually without being interrupted, for a typical recording session lasts 20~30 minutes. The app cannot be suspended, killed, or switched to another app. It must be secure from interruptions while running. For multiple recordings, the app must be operational for long periods of continuous usage without any restart or crashes. The app is data-intensive in that it needs to transmit a large volume of

EEG samples and process them in near real-time. It devotes most of its processing time to network I/O and manipulation of the data. Each 20~30 minutes recording session produces about 2 GB of EEG samples, and at its maximum sampling rate it requires 0.96 Mbps data transfer rate between the device and the app (see Section 3.1).

3.1. Data Model

A HifoCap device consists of 24 electrodes, small metal discs with thin wires, to record the electrical signals of the brain and send them to the HifoCap app. Each signal recorded from an electrode is digitized into a 16-bit integer value. The HifoCap device supports a wide range of sampling rates from 250 to 2500 samples per second. The class diagram in Figure 4 defines three key data entities and their relationships.

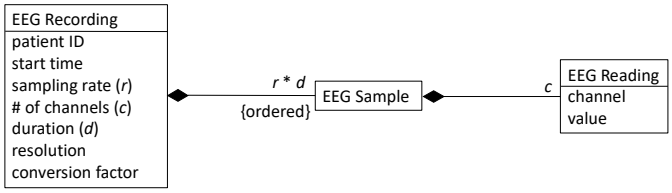

Figure 4. Data model

An *EEG recording* is a sequence of *EEG samples* of one EEG time series. It is the unit of storing and transmitting EEG samples to the server, containing metadata such as patient ID, start time, sampling rate (in samples per second), duration (in seconds), resolution (in bytes per channel), and conversion factor (from sample values to microvolts). An *EEG sample* is a set of EEG readings, one per channel, read together at a specific time. An *EEG reading* gives the channel and the electrical signal read at the channel. The signal strength is specified in microvolts (μV). A typical recording configuration consists of 24 channels and 2500 samples per second.

3.2. Challenges

There are many challenges in developing the HifoCap app on the Android platform. One challenge like other mobile app developments today is the diversity of devices and platforms [7]. There is a multitude of devices, ranging from watches to smartphones, and to tablets, each with different screen sizes, operating systems, and other characteristics and capabilities. It is a real challenge to be able to create an app that would run smoothly across devices and platforms. However, the most unique and interesting challenge for the HifoCap app is to ensure the soft real-time requirement on non-dedicated Android devices where uncertainty and unpredictability are the norms, not the exception.

- During execution, the app may face several interruptions like incoming calls, text messages, and various notifications.
- While the app is running, another app may be launched by the user or the Android operating system.
- There may be many background services running concurrently with the app.

- The app doesn't even have full control of its own life cycle. It may be killed, suspended, or deprioritized by the Android system relative to other apps and services.
- When the available memory is low, the Android garbage collection is initiated automatically, potentially slowing down the app.
- The network signal strength may become weak causing network coverage outage.

All of the above also affect the performance of the app. As a data and network-intensive app, the HifoCap app has to process a large volume of data, e.g., 2500 samples per second at the highest sampling rate, requiring 0.96 Mbps data transfer rate between the HifoCap device and the app. It is well-known that the performance can be degraded severely under certain conditions such as low battery, bad network coverage, and low available memory. Thus, identifying the app's performance bottlenecks and addressing them is critical to the success of the app [8]. Minimizing the garbage collection execution time is also important because garbage collection in general results in poor performance of the app and the overall slowdown of the system; the app may be suspended during garbage collection. The battery consumption is also an important concern, and the challenge is to design a well-performing app which runs on a minimum of power consumption. As mentioned in the previous section, a typical EEG recording session lasts 20~30 minutes and the app should support multiple, continuous sessions. The above-mentioned factors also affect the long and continuous operation of the app. In particular, the app may be killed, suspended, or deprioritized relative to other apps and services. For example, the smartphone or tablet can go to sleep by turning off the screen or a user may accidently launch another app causing the HifoCap app to be paused, stopped or even destroyed. It is not straightforward to design an app to run and operate efficiently for a long period of time. As the HifoCap app needs to be continually running, there is no room for memory leaks or anomalies that may lead to a crash. One non-technical challenge is that there is no HifoCap device available. It is to be developed by a separate, hardware team.

4. Design and Implementation

In order to address the challenges mentioned in the previous section and to meet the performance requirement as well, we came up with several different solutions including:

- Reduce potential interruptions.
- Minimize garbage collection time.
- Minimize network and I/O time.
- Visualize samples selectively.
- Design a light-weight UI.
- Create a HifoCap device emulator.

As described previously, an app may face different interruptions like incoming calls, text messages, and notifications, and it can be suspended or even killed by the Android operating system. We can reduce these potential interruptions and disruptions by temporarily disabling other apps, background services, and any hardware features not required by the HifoCap app before starting EEG recording sessions and enabling or restoring them when the recording sessions end. It may be even suggested to remove

non-relevant apps that could take up a significant processing time. The goal is to ensure a sort of singular control of the device by the app, thus approximating a single-tasking environment, so that it can be continually active. Ideally, the app shouldn't be forced to terminate a recording session due to unrelated external interruptions.

The Android garbage collection is unpredictable in that it can be initiated at any time during the execution of an app to automatically manage the memory used by the app. In general, garbage collection has a negative impact on the response time and the stability of an app, and it can be optimized in such a way that its impact on the app's response time or CPU usage is reduced and minimized. There are many performance tips for minimizing garbage collection execution time, e.g., avoiding creation and destruction of unnecessary objects and managing some of the required memory by using object pooling and sharing to support a large number of little objects efficiently [8] [9] [12]. The use of managed object pools will definitely reduce the defragmentation of memory and the pileup of unreferenced objects that force garbage collection.

The HifoCap app is network and I/O-intensive in that a large volume of data must be received at 0.96 Mbps and about 2GB of EEG samples must be transmitted to the server periodically, e.g., at the end of each recording session. Network and I/O operations are significantly slow and expensive compared to computation. The standard technique for improving I/O performance is to use buffering. We use buffers to accumulate and temporarily store EEG samples before transmitting them to the network or I/O system (in the case of writes) or before providing them to the consumers (in the case of reads). By buffering the EEG samples, we can reduce the number of I/O operations and improve the overall performance significantly.

The HifoCap app is a data-intensive, soft real-time app in that it has to process a large volume of EEG samples in near real-time, e.g., 2500 samples per second at the highest sampling rate with each sample consisting of 24 EEG readings, one for each channel. We doubt that most Android devices will be able to decode and visualize EEG samples at that rate. In fact, the screen refresh rate of most Android devices is 60 Hz; the Android system refreshes the screen at most 60 times per second. Our approach is to visualize EEG samples selectively, to select 1 out of every n samples. This reduces the execution time needed for visualizing samples as well as decoding them, as samples can be dumped to a file in the raw data format for later uploading to the server.

The design of UI can be a determining factor in meeting the soft real-time requirement. The app visualizes EEG samples by plotting them lively to allow for live monitoring of real-time EEG samples. It might be time and memory-intensive to update the displayed graphs in real-time as EEG samples are received. The standard UI design pattern for Android is to use multiple activities and multiple views. An *activity* is a unit of Android programs responsible for a single screen. If an app consists of two screens, the standard approach is to create two activities, each launched separately and having a separate life cycle. However, this approach incurs significant heap and context switching overhead because each activity is a separate task. For more efficient and responsive UI, we can create a light-weight UI consisting of a single activity but multiple views. The performance of a UI is also affected by its layout, and a common rule of thumb when choosing layouts is to select a combination that results in the smallest number of nested layout views. The performance is generally better if one flattens the layout or making it shallow and wide, rather than narrow and deep.

We created an app to emulate the HifoCap device to be constructed by the hardware team. The emulator app running on another Android device mimics the HifoCap device by generating EEG samples and sending the generated samples to the HifoCap app

through a Bluetooth connection. It also accepts and responds to control commands from the HifoCap app to configure itself and manage recording sessions. The emulator app lets us test our code as if interacting with a real device. It is a key tool to test various conditions and situations that are difficult or impossible to duplicate with a real device. It also allows us to perform various experiments to meet the app performance and soft real-time requirements.

4.1. Design

In addition to the requirements and the challenges discussed earlier, we also consider extensibility and flexibility to be important in the design of the HifoCap app. We need to provide for change while minimizing impact to existing components because of several reasons, including:

- To integrate easily with other components of the HifoCap system, such as the HifoCap device and the HifoCap cloud server, that are developed by different teams. The interfaces and communication protocols need to be worked out and are likely to evolve or be refined as the development progresses.
- To facilitate a variety of experiments and feasibility studies. To address standard requirements and constraints from the domain of mobile health devices and apps, we plan to perform various types of experiments and feasibility studies especially on the performance and stability of the app. For example, there will be feasibility studies for additional features such as real-time analysis of EEG samples.
- To support an iterative development. We will develop the app incrementally in small steps to receive feedback early and frequently from users and reviewers with expertise in biomedical engineering and healthcare delivery.

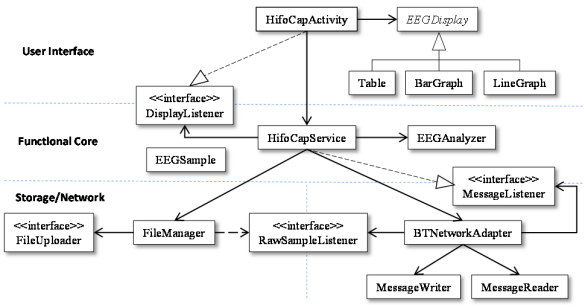

Figure 5. Class diagram

Figure 5 shows main classes of the HifoCap app along with their relationships such as associations and inheritance relationships. As shown, our design uses a layered architecture consisting of three layers: the user interface layer, the functional core layer, and the storage and network layer. The UI layer interacts with the user and provides three different ways for visualizing sampled EEG signals (see Section 4.2 for sample screenshots). It obtains EEG samples from the functional core layer through the *DisplayListener* interface. The functional core layer is responsible for managing the EEG sampling and recording sessions by configuring the HifoCap device and starting and stopping the sessions. The key class is the *HifoCapService* class which sends appropriate

messages to and receives incoming messages from the device through the storage and network layer. It is an Android *service*, an application component that doesn't have a user interface and performs long-running operations typically in the background. The storage and network layer is responsible for communicating with the HifoCap device, storing received EEG samples temporarily in a local storage, and uploading them to the cloud storage server periodically. The *BTNetworkAdapter* class from the network layer is fully responsible for networking and communicating with the device. It encapsulates network details such as protocols, message formats, and endianness from the rest of the system. The class consists of two active classes, each with its own thread, to send and receive messages asynchronously. Upon receiving EEG samples, the class passes them without decoding to the *FileManager* class so as to store and upload them to the server. If a received sample needs to be visualized or analyzed, it is passed to the *HifoCapService* class in the functional core layer via the *MessageListener* interface.

It would be instructive to see how our design meets the extensibility requirement. We assign different responsibilities to different layers and modules and separate them cleanly. We apply the principles of the strictly layered architecture, especially no dependency from a lower layer to an upper layer. To define module boundaries cleanly and remove unnecessary dependencies among modules, we use Java interfaces, the Observer pattern [10], and other well-known design principles such as the *dependency inversion principle* [11] demanding to depend upon abstraction, not on concretions. Lower-layer modules, for example, communicate with upper-layer modules only through well-defined interfaces such as *DisplayListener* and *MessageListener*. It is also the case for reporting errors and other status information to the UI layer. We also use the Strategy pattern [10] for extension in two different places, displaying EEG samples (*EEGDisplay*) and uploading EEG samples to the cloud server (*FileUpLoader*). Although the class diagram doesn't show the details, the performance was an important driving force for our design to meet the soft real-time requirement. The UI consists of a single activity class, *HifoCapActivity*, in pursuit of a lightweight UI. We use multithreading making several classes such as *HifoCapService*, *FileManager*, *MessageWriter*, and *MessageReader* active in that they have their own threads of control, and this improves both the performance and the responsiveness of the app (see Section 5).

4.2. Implementation

Figure 6 shows sample screenshots of the HifoCap app. As shown in the first screenshot, the steps for connecting the HifoCap device are the same as those for other Bluetooth devices. Once the device is connected, one can configure it by setting options such as channels and sampling frequency and then tapping the *Configure* button. Once configured, the device can be instructed to sample and transmit EEG signals by tapping the *Start* button. As shown, samples can be displayed in a few different ways, and the display rate can be changed dynamically.

The implementation of the HifoCap app consists of 2569 lines of Java source code, not including various XML resources such as layouts, menus, values, and styles. It consists of 9 interfaces, 72 classes (including 36 anonymous classes), 2 enums, and 386 methods; the named classes include 17 nested classes. The implementation is mostly a direct translation of its design. However, one implementation-level refinement was to get rid of record-like model classes such as *EEGRecording*, *EEGSample*, and *EEGReading* from the functional core layer. The use of these classes requires, for each decoded EEG sample, to create one instance of the *EEGSample* class and up to 24

instances of the *EEGReading* class. It would make our app memory-hungry by creating millions of little objects and forcing a garbage collection every few seconds (see Section 5). It would have a huge negative impact on the performance and stability of the app. Allocating memory is always more expensive than not allocating. We instead used *short* arrays to store decoded samples; each EEG reading is a 16-bit integer.

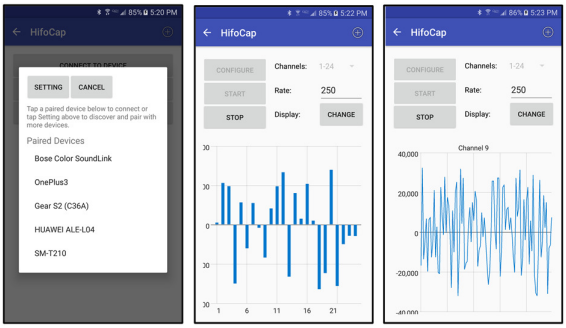

Figure 6. Sample screenshots

In the detailed design and implementation, we used well-known design principles such as SOLID principles [11] and design patterns [10] judiciously. For example, we introduced interfaces to clearly define module boundaries and be explicit about the required/provided interfaces of modules, and all interfaces have only a few methods. The guiding principle was the *interface segregation principle* stating that "many client-specific interfaces are better than one general-purpose interface" [11]. However, we limited the total number of interfaces by carefully selecting the places to introduce interfaces because they might have a negative impact on the performance, e.g., the overhead of dynamic method dispatch in the presence of multiple concrete classes. Uploading EEG samples to the cloud server is an important functionality, however, there was a concern in its implementation. The server was being developed and configured by another team, and its details including the platform and protocols were not known. We used the Strategy design pattern [10] by defining an interface *FileUploader* declaring an *upload*(*FileInfo*) method. This method is called by the *FileManager* class when the EEG samples, temporarily stored in a local file, meet the uploading criteria, e.g., file size. With this framework in place, we were able to test our design and implementation by creating an example concrete strategy, *DropBoxUploader*, to upload EEG samples to Dropbox. Throughout our implementation, we also paid special attention to the performance and efficiency of our app by using several profiling tools (see Section 5). As discussed before, we wrote a HifoCap emulator, an Android app consisting of 708 lines of Java source code. It was invaluable not only for testing our app but also for performing various performance-related experiments. One side benefit was capturing and documenting explicitly all the assumptions that we made about the HifoCap device including communication protocols in a single place, the *DeviceProfile* class, which is shared between the app and the emulator. The emulator also confirmed quickly that the Bluetooth Low Energy protocol doesn't provide an adequate data transfer rate for us; we initially considered its use for its considerably reduced power consumption especially on the HifoCap device.

5. Evaluation

We measured the time and space efficiency of our app to evaluate quantitatively the effectiveness of our approach. It wasn't straightforward to measure the execution time in the presence of multithreading and a multicore processor, and thus we measured different types of execution times such as displaying, processing, and I/O separately. The smartphone for the HifoCap app has a 1.5GHz octa-core processor with 2GB RAM and 16GB internal storage and has Android 5.1. The smartphone for the emulator runs Android 6.0.1 and has a 2.15GHz/1.6GHz quad-core processor with 4GB RAM and 32GB internal storage. Below we first show some of the evaluations that we performed for the execution time efficiency.

To compare the impact of buffering in transmitting EEG samples over Bluetooth, we measured the execution time needed to receive and process 10,000 EEG signal samples at different sampling rates and averaged 1,000 measurements. The processing time doesn't include the display time used by the UI thread (see below). We also measured the numbers of read operations and the numbers of bytes read per read operation, which might affect the execution time. The measurements are summarized in Table 1. For the buffered I/O, we used the default buffer size of the Android OS.

Table 1. Unbuffered and buffered I/O, where SR: sampling rate (samples/sec); TT: total time (msec), RT: I/O time (msec), NR: # of read calls, NB: # of bytes read per read call.

SR	Unbuffered				Buffered			
	TT	RT	NR	NB	TT	RT	NR	NB
250	53105	53097	7010	72	48770	48755	713	674
500	30121	29988	5381	96	27643	27502	714	680
1000	17522	17480	3552	140	15710	15669	713	688
1500	12804	12791	2342	172	10843	10829	711	698
2000	10127	10105	1493	207	7843	7822	689	714
2500	8650	8636	1215	245	5812	5797	564	868
Avg.	22055	22016	3499	155	19437	19396	684	720

An immediate observation is that most of the execution times (TT; 99.73%~99.97%) are spent on reading incoming messages (RT) for both the unbuffered and the buffered I/O, meaning that the app is I/O intensive. The average number of bytes read per I/O operation (NB) is 155 for the unbuffered and 720 for the buffered. As a result, there are significant differences in the numbers of read operations (NR) needed between the two, on average 3499 vs. 684 calls, as depicted in Figure 7. These differences translate to the overall and I/O performance improvements (TT and RT; 8%~33%) of the buffered I/O as depicted in Figure 8. However, there are two interesting points to note. First, as shown in the graph, the overall performance improvements are the same as those of I/O improvements. Second, even if there are significant differences in the numbers of read operations at lower sampling rates, 250-1000, the performance gains are relatively small compared to those of higher sampling rates (1500-2500). The gains in the numbers of read operations might be compensated by the blocked I/O operations waiting for the buffer to be filled up. Another finding is that if the emulator doesn't use buffering, the result is similar to the unbuffered case, therefore it's essential for the HifoCap device to use buffered I/O to transmit EEG signal samples.

We measured the execution time required by the UI thread to display EEG samples. For this, we measured the time needed to display 10,000 EEG signal samples at the highest sampling rate (2500) for all 24 channels and averaged them over 1000 measurements (see Table 2).

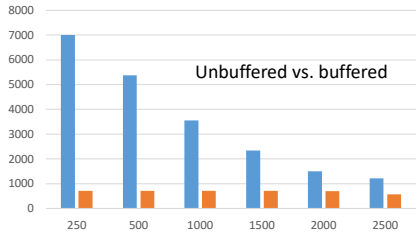

Figure 7. Number of read operations

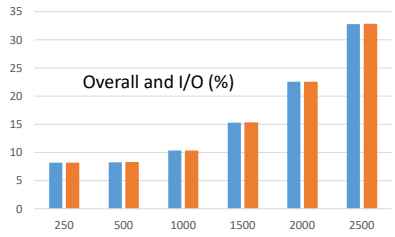

Figure 8. Performance gains of buffered I/O

Table 2. Display time

Type	Execution Time (msec)		
	Display (D)	Proc (P)	% (D/P)
Table	232	5909	4
Bar graph	2243	5843	28
Line graph	4028	5917	41

As expected, the table display that displays samples in a table has the least overhead while the line graph display has the most overhead at 41%. This indicates that selective display is more effective for graph displays. Indeed, for the graph displays the performance gains are incomparable. We believe that the main contributing factors to these improvements are dynamic memory allocation and garbage collection (see below). Prior to this measurement we also expected selective decoding—decoding only those samples that need to be displayed—to have a similar performance gain. However, it wasn't the case because, as shown earlier, more than 99% of processing time was used by I/O to receive EEG samples and less than 1% for decoding them. The app is I/O intensive, not computation intensive.

We measured the effect of getting rid of model classes such as *EEGSample* and *EEGReading* in our implementation. These are record-like classes with no significant behavior. Our motivation was to reduce the garbage collection execution time; they would produce a huge number of little objects rapidly because each EEG sample needs one instance of *EEGSample* and 24 instances of *EEGReading*. We measured the execution time required for processing 10,000 EEG signal samples at 2500 samples per second for all 24 channels and averaged 1000 measurements. The results are shown in Table 3. The first column is the decoding rate, the number of samples decoded per second. The improvements are consistent at average 10.5%, which is significant considering that only two classes are removed.

Table 3. Object vs. array

Decoding rate	Execution time (msec)		
	Object	short []	Gain (%)
0.25	7330	6429	12
0.5	7282	6505	11
1	7277	6677	8
2	7266	6448	11
4	7327	6524	11
all	7297	6542	10

Table 4. Dynamic memory allocation

Memory Allocation (%)		
I/O Thread	UI Thread	GC Interval
96%	4%	3m11s

We also examined the space efficiency of our app. Memory allocation is an important factor that affects the performance of an app on Android because it may trigger garbage collection. We used profiling tools to study the memory allocation pattern of our app. We observed the memory allocation of our app while sampling EEG signals at 2500 samples per second for all 24 channels and displaying 1 sample per second. We learned that two threads allocate most memory: the I/O thread responsible for receiving and decoding incoming messages from the emulator and the UI thread (see Table 4).

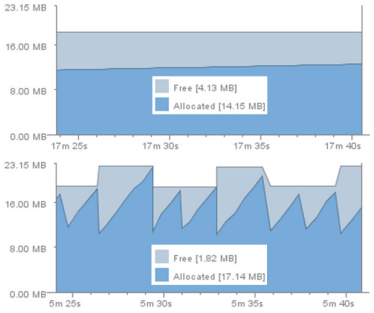

Figure 9. Memory allocation when displaying one sample per second (top) and every sample (bottom)

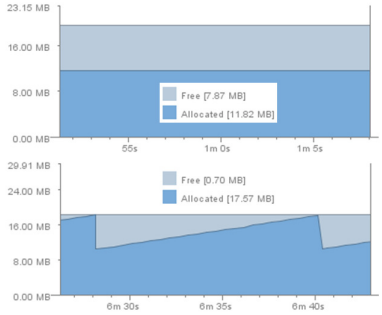

Figure 10. Memory allocation with managed object pools

An interesting finding is that the memory consumption pattern changes radically depending on the number of samples displayed per second (see Figure 9). When one sample is displayed per second, memory consumption increases in a linear fashion with an average garbage collection interval of 3 minutes and 11 seconds. However, when every sample is displayed, it becomes very dynamic and lively. It not only forces garbage collection at every few seconds but also make the UI thread consume more memory (UI: 96% and I/O: 4%). The app becomes display-intensive. This confirms that it was indeed a good idea to visualize samples selectively.

Recall that one of our solutions to the challenges is to minimize garbage collection execution time. As described above we learned that the I/O thread (*BTNetworkAdapte*r) is responsible for most of the dynamically allocated memory. It was no surprise because it creates large numbers of objects: (a) *byte* arrays to dump raw samples asynchronously to a local file and (b) *short* arrays for decoding and displaying samples selectively. We reduced the number of object creations and destructions by managing the required memory for these two cases; we used object pooling and sharing to support a large number of little objects efficiently by applying the Flyweight design pattern [1]. As depicted in Figure 10, sharing objects through the use of managed object pools produced an astonishing result. The dynamic memory allocation is almost unnoticeable when one sample is displayed per second (top graph), requiring no garbage collection over a long period of time; the line is flat and, though not shown, it also changed the memory allocation pattern completely to 0% for I/O and 98% for UI. It also reduced the number of garbage collections to 60% when every sample is displayed (bottom). After this optimization, the app became display-intensive from a memory usage point of view.

We also applied a similar technique to the UI thread to optimize its dynamic memory use. We avoided dynamic creation of display-related objects and, if possible, reused them. For this, we often had to introduce subclasses of the UI framework classes to make their instances reusable by providing mutation methods. However, there was a serious

obstacle in the use of managed object pools for UI. The Android standard and open-source UI classes were responsible for most of the memory allocation (95% in size; see Figure 11), and their methods were called indirectly by our code and their internal structures were not accessible. Unless these UI framework classes are rewritten, which would be a daunting task, the use of object pools will have a limited impact on the memory efficiency.

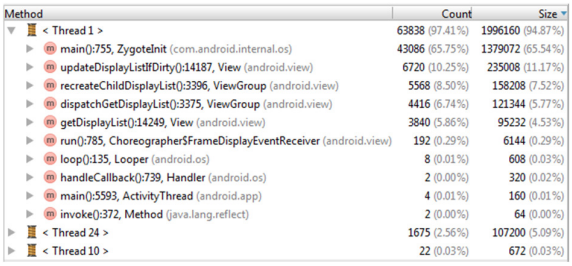

Figure 11. Memory allocation by UI (Thread1)

The primary use of our app is to record and visualize EEG signals in a near real-time fashion. To test this, we measured the time needed to transmit EEG samples from the emulator to the app's device and then to display them on the screen. We called it a *delay time.* It includes the transmission time, processing time, and display time, and in a sense is a response time that a user perceives. For this measurement, we generated EEG samples at the maximum sampling rate (2500) for all 24 channels and displayed all samples without dropping any. On the emulator, we time-stamped every 10,000th sample, that we called a *tracer bullet sample*, just before transmission. On the app, upon displaying a received sample we checked if it was a tracer bullet. We time-stamped each tracer bullet sample right after displaying it and calculated its delay time. The delay times of all measured tracer bullet samples were in the range of 32~233 milliseconds with an average delay of 92 milliseconds. This is good and acceptable for our app. In fact, we can see it visually using the tabular display on the app; the emulator and the app display EEG samples almost simultaneously. We were also able to run our app for many hours without any problems. However, we noticed that the smartphone running the emulator gets somewhat hot after a few hours, perhaps because of a high data transfer rate. It would be a concern to the HifoCap device if it is indeed caused by a high data transfer rate.

6. Conclusion

The hardware capability and available sensors on mobile devices such as smartphones and tablets enable use cases that were previously unimaginable, e.g., various types of wearable mobile health devices and apps. We have been developing one such an app on the Android platform, the HifoCap app to receive electroencephalogram (EEG) signal samples from a wearable device hidden in a cap, to visualize the received samples, and to transmit them to a cloud storage server for off-line analyses. We identified many interesting challenges in developing a data-intensive, soft real-time app on non-dedicated Android devices such as smartphones and tablets. We grouped these challenges by their causes, including interruptions (such as incoming calls, text messages, and notifications),

no control on an app's lifecycle, unpredictable garbage collection, high network bandwidth utilization, long and continuous running, and network coverage outage. We proposed possible solutions to some of these challenges—e.g., reducing potential interruptions, minimizing the garbage collection execution time, minimizing the network and I/O time, selective visualization, and light-weight UI—and implemented them in our app. We measured the time and space efficiency of our app and evaluated the effectiveness of our proposed solutions quantitatively. All our solutions were effective and obtained performance gains in the range of 8%-257%, and object pooling nearly eliminated the need for garbage collection in typical use of the app. The app was able to receive and visualize EEG samples in near real-time; the average delay time between EEG sampling on the device and visualization on the app is 92 milliseconds. We believe that our proposed solutions are applicable to other soft real-time apps targeted for non-dedicated Android devices. However, one remark is that one has to pay special attention to performance throughout the development, especially during coding. It isn't uncommon to learn that one simple, innocent line, e.g., using *ByteBuffer.wrap* to decode samples, costs one dearly.

Acknowledgements

This work was supported in part by the Interdisciplinary Research Fund, College of Engineering, The University of Texas at El Paso (2015-2016). Thanks to Oliver Singayigaya for eliciting the requirements and John Ramirez for helping in the initial prototype implementation.

References

[1] H. Seabrook, et al., Medical applications: a database and characterization of apps in Apple iOS and Android platforms, *BMC Research Notes*, August 2014. DOI: 10.1186/1756-0500-7-573.

[2] L. Perneel, H. Fayyad-Kazan, and M. Timmerman, Can Android be used for real-time purposes, *Computer Systems and Industrial Informatics*, December 18-20, 2012. DOI: 10.1109/ICCSII.2012.6454350.

[3] E Niedermeyer and F. da Silva, Electroencephalography: Basic Principles, Clinical Applications, and Related Fields, Lippincott Williams & Wilkins, 2004.

[4] N. van Klink, et al., High-frequency oscillations in intraoperative electrocorticography before and after epilepsy surgery, *Clinical Neurophysiology*, 125(11), March 2014. DOI: 10.1016/j.clinph.2014.03.004.

[5] W. Sato, et al., Rapid, high-frequency, and theta-coupled gamma oscillations in the inferior occipital gyrus during face processing. *Cortex,* 60: 52-68, November 2014.

[6] R. Zelmann, Scalp EEG is not a blur: it can see high-frequency oscillations although their generators are small, *Brain topography* 27(5): 683-704, September 2014.

[7] M. Joorabchi, A. Mesbah and P. Kruchten, Real challenges in mobile app development, *International Symposium on Empirical Software Engineering and Measurement*, October 2013, pages 15-24.

[8] M. Linares-Vasquez, et al., How developers detect and fix performance bottlenecks in Android apps, *IEEE International Conference on Software Maintenance and Evolution*, September 2015, pp 352-361.

[9] Y. Cheon, *Are Java Programming Best Practices Also Best Practices for Android?* Technical Report 16-76, Dept. of Computer Science, University of Texas at El Paso, El Paso, TX, October 2016.

[10] E. Gamma, et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[11] R. Martin, *Agile Software Development, Principles, Patterns, and Practices*, Pearson, 2002.

[12] D. Sillars. *High Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing*, O'Reilly, 2015.