Sharing is Caring: A Data Exchange Framework for Colocated Mobile Apps

by

Joseph Milazzo

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2014 by the
Graduate Supervisory Committee:

Sandeep Gupta, Chair
Georgios Varsamopoulos
Brian Nelson

ARIZONA STATE UNIVERSITY

May 2014

ABSTRACT

Mobile apps have improved human lifestyle in various aspects ranging from instant messaging to tele-health. In the current app development paradigm, apps are being developed individually and agnostic of each other. The goal of this thesis is to allow a new world where multiple apps communicate with each other to achieve synergistic benefits. To enable integration between apps, manual communication between developers is needed, which can be problematic on many levels. In order to promote app integration, a systematic approach towards data sharing between multiple apps is essential. However, current approaches to app integration require large code modifications to reap the benefits of shared data such as requiring developers to provide APIs or use large, invasive middlewares. In this thesis, a data sharing framework was developed providing a non-invasive interface between mobile apps for data sharing and integration. A separate app acts as a registry to allow apps to register database tables to be shared and query this information. Two health monitoring apps were developed to evaluate the sharing framework and different methods of data integration between apps to promote synergistic feedback. The health monitoring apps have shown non-invasive solutions can provide data sharing functionality without large code modifications and manual communication between developers.

This is dedicated to my loving and supporting mother.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Sandeep K.S. Gupta for giving me many opportunities and encouragement at Impact Lab. I would like to thank my committee members Dr. Brian Nelson and Dr. Georgios Varsamopoulos for their valuable suggestions during my thesis. I would like to extend a great thanks to Dr. Ayan Banerjee and Priyanka Bagade for helping me so much in my thesis work. I would also like to thank Impact Lab members for making the lab like a family, including but not limited to Zahra Abbasi, Sunit Verma, Joshua Ferguson, Madhurima Pore, and Koosha Sadeghi. I thank National Science Foundation (through grants IIS-1116385 and CNS-0831544). I would also like to thank my sister for proofreading so many papers for me and my family and friends for their love and support.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Imagine a world where your workout app is aware of cheat days and adjusts accordingly without extra input; imagine if completion of a simple exercise in your workout awarded you points that could be used to unlock characters in other apps like the casual game Angry Birds; imagine if downloading one app allowed for integration of your favorite apps.

In order for this dream to become reality, mobile apps need a way to communicate and share data with each other. This shared data needs to be integrated in a way that reduces user intervention and motivates inter-app use.

Unfortunately, at the current state, data sharing is not easily achievable. If one-way data sharing between two applications is to take place, either one of two approaches is taken: the developers contact each other and share internal details of data storage; or the developer of the app whose data is being read must develop a custom Application Programming Interface (API) for other apps to access data. This places a burden on the developer to develop and maintain such an API.

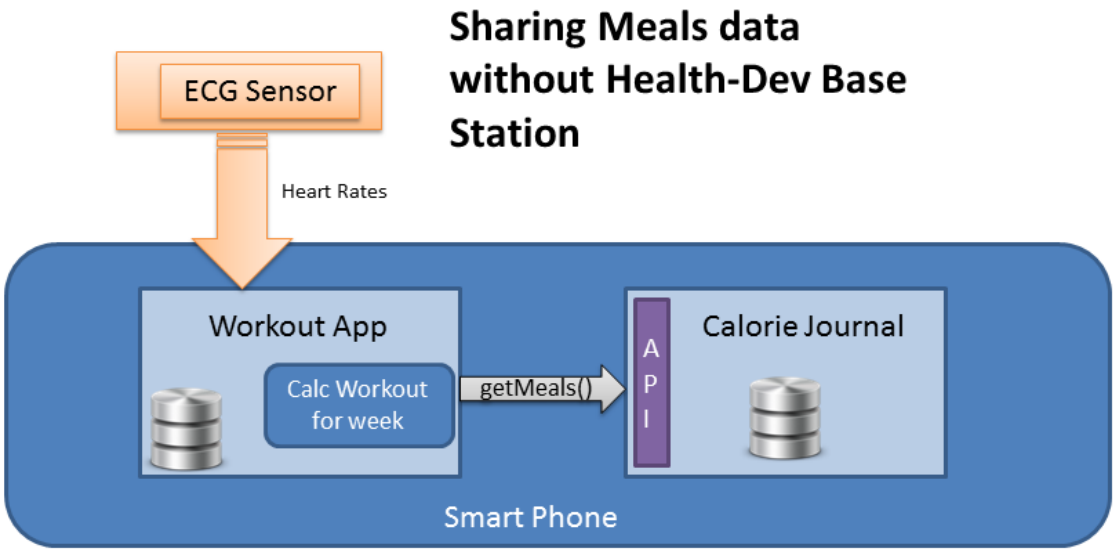

Figure 1.1: System model of sharing meals table between apps without framework.

The framework proposed in this thesis attempts to be a simple approach to app integration while non-invasive to the developer. Instead of writing a custom API, the developer only needs to register tables of their database before another app can query details from the framework to form a query for the registered app's database. Figures 1.1 and 1.2 illustrate how the framework affects data-sharing. The two apps shown are Workout App, used to plan a workout schedule based on a user-defined goal; and Calorie Journal, a calorie intake diary to track diet and calorie/nutrient needs. Without the framework (Figure 1.1), a custom API is developed for Calorie Journal and thus Workout App can use the API to access the database. However, with the framework, Calorie Journal first registers a table with the framework; Workout App can then query the framework to learn which tables Calorie Journal supports allowing it to easily query Calorie Journal's database. One interesting difference between the two systems and that of the framework is that the external sensor is communicating only with Base Station (Sharing and Caring Framework) rather than having to also

communicate with Calorie Journal. A benefit to this is the sensor data may be shared efficiently among multiple apps in real-time. In addition to data sharing, the framework can interface between sensors and apps. With the Base Station handling the communication between sensor and app, the possibility of another app making use of the sensor's data exists and leads to shared functionality (Chapter 4).

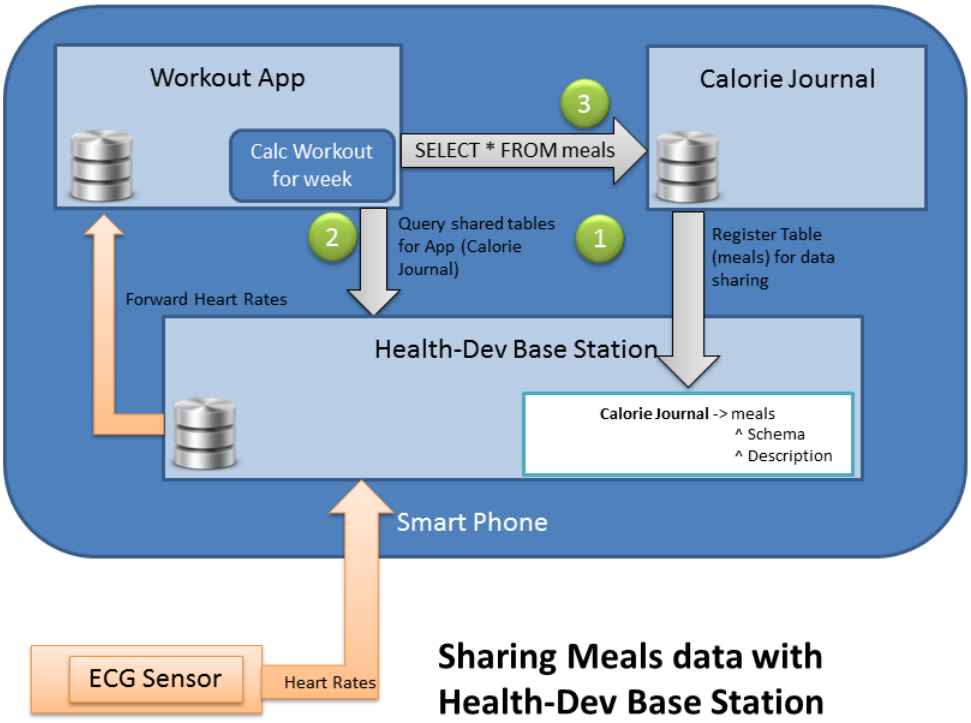

Figure 1.2: System model of sharing meals table between apps with framework.

This thesis describes the creation of a framework for sharing data between mobile apps non-invasively and interfacing between sensor and multiple apps. Non-invasive in this context refers to minimizing required changes made in an app to achieve shared data. The framework allows multiple apps to become aware of users' activities and physiological data through shared data from other apps in only a few steps. To validate the framework, this thesis introduces a suite of apps designed to share data for

multimodal feedback and motivate usage of multiple behavioural health monitoring apps. Lastly, this thesis presents multimodal feedback techniques discovered through the design of the app suite and discusses their effectiveness in motivating inter-app use. Multimodal feedback is a form of feedback which utilizes multiple forms of data. For example, multimodal feedback may take the form of visual explosions on a screen; with explosions appearing more frequently the more relaxed the user is. Relaxation is calculated based on the brain waves and breathing rhythms of the user. The feedback is multimodal because it is derived from two types of data, brain waves and breathing rhythms, being used together in one feedback loop.

1.1 Objective

The objective of the thesis is **to facilitate sharing of processed data between multiple mobile apps**.

1.2 Challenges

Achievement of this objective poses several challenges (outlined below) that need to be addressed.

1. Currently for two apps on the same system to share data, the developer must develop an API to grant access to the shared data. If multiple apps wish to share data, each app will need to develop an API. With so many APIs the question arises: **How to enable data sharing, bypassing collaboration between developers?**
2. **How to give simultaneous sensor data access to multiple apps?**
3. **How to develop a solution that addresses the challenges without performance degradation?**

1.3 Contributions

Main contributions of the thesis are summarized below:

1. A methodology to interface shared data access between apps;
2. A suite of health monitoring apps utilizing shared data for multimodal feedback;
3. A service to interface sensor and app which can route sensor data to multiple apps.

1.4 Solution Approach

The goal of this thesis is to provide integration of apps in a non-invasive way and minimize the additional work on the developers. This thesis chose to separate interfacing from development and hypothesizes that the separation of interfacing from the developer results in easier app implementation.

Interfacing is classified into two aspects: 1) sensor interfacing, which is connecting and communicating with a sensor and 2) inter-app interfacing, where two apps share inferences from processed signals. For sensor interfacing, automatic code generation was used to generate the interface code between apps and a sensor. There are many automatic code generators available. One is Health-Dev (see Chapter 4), to which I contributed. Code generation reduces human effort and error while giving greater control over how communication between smart phone and sensor works. In addition, it contributes to an increase in quality of code reducing common development pitfalls such as index out-of-bounds exceptions, infinite loops, and deadlocks. Inter-app interfacing is the sharing of data between two apps such that the shared data may have an effect on the other app to give synergistic feedback to the user.

By separating the interfacing from the developer, the solution is non-invasive which means faster development because sensor development, Bluetooth, sensor safety and sustainability (Gupta *et al.* (2013)), and other domain specific knowledge is not needed from the developer. For example, if a developer wished to connect to an external ECG sensor to an app and calculate user heart rate a developer would need to develop sensor code and the communication between the sensor and the app code. Instead, relying on Health-Dev for code generation, a developer can quickly generate code for a sensor and the needed code to communicate with said sensor. This allows leaving the developer to focus on user interface and processing algorithms; if they want a sensor, to communicate with the cloud, store data or access from other apps, no code needs to be touched. Instead, the developer can rely on Health-Dev to generate the functionality needed.

Chapter 2

RELATED WORKS

This chapter discusses the current state of data sharing in mobile apps and the limitations these place on the advancement of data sharing in a plug-n-play system. This chapter first reviews the current state of inter-app communication across iOS and Android, two popular mobile operating systems. This chapter then discusses the perceived limitations of the current state and compares related works against different granularity of changes needed.

2.1 Preliminaries

2.1.1 iOS

iOS runs apps in sandboxes which provide limited means of communication between apps directly on the device. Shared files and messaging systems are currently not present in iOS. However, iOS does provide the ability for apps to register URL Schemes which are currently used by many apps to launch other apps and pass basic data through URL parameters. However, there is no structure to these URLs and no current standard for allowing callbacks to be passed in the URL if the originating app wishes to receive some result based on the action.

Vivo (2013) proposes an Open Source library, Inter-App Communication, to improve on iOS lack of inter-app communication by providing callbacks to URL Schemes. This framework allows for callbacks to be registered based on the x-callback-url protocol specification. These callbacks allow an app to develop an API for data sharing that can return results to the requesting app.

2.1.2 Android

Android also sandboxes apps, however it provides ample resources for inter-app communication. In Android, apps can register themselves to handle specific media types in an app, such as opening a file type; uni- or multi-casting data between other apps on the phone; granting read/write permission on their database. Android already supports shared functionality by allowing apps to register themselves as handlers for opening, sending, viewing different forms of media.

In order for two apps to communicate in Android, a broadcast receiver and a broadcast must exist. A broadcast is a structured message for sending data across the Operating System and a broadcast receiver listens for different broadcasts. This makes creating an API for data sharing easily feasible since broadcasts can also specify order in which they are delivered to different apps and if they are run asynchronously or not.

To expand data sharing natively to Android, an app would need to define permissions for each app that wishes to access the database or expose their schema. As discussed in the introduction, exposing schema is not a feasible option and defining permissions limits the plug-n-play functionality in data sharing. The literature below, suggests that providing an interface is an appropriate solution. An interface allows the developer to easily gain some functionality without worrying about the technical details.

2.2 Data Sharing Interface

Chun *et al.* (2012) presents Mobius, a middleware for interfacing with complex data management. Mobius provides programming abstractions of logical tables of data that spans devices and clouds. Applications using Mobius can asynchronously

Table 2.1: Comparison of different interfaces in mobile systems.

Interfaces	**API**	**Structural Changes**	**Interoperable**	**Code Generated**
Mobius	Required	Minimal	No	No
Simba	Required	Minimal	No	No
SOCAM	Required	Large	No	No
Data Sharing Interface	Optional	None	Yes	Yes

read and write these tables and receive notifications when tables change via continuous queries on the tables. The developer has a problem of complex handling of data management. There is no native solution from the platform, thus Mobius provides an interface to simplify data management. Another interface (Simba) is proposed by Agrawal *et al.* (2013) which is simplifies the complexities of synchronizing data to the cloud with minimal work from the developer through a single interface. Gu *et al.* (2004) proposes a Service-Oriented Context-Aware Middleware (SOCAM) to enable rapid prototyping of context-aware services in pervasive computing environments. SOCAM provides a middleware of components to define context providers, interpreters, and the interaction between different components. The middleware can allow data to be interpreted differently between multiple apps; however each app must use the middleware to participate.

Table 2.1 provides an overview of the different interface approaches taken over the literature discussed. As the table describes, many solutions use an API to provide the interface and require an app to change structural design to adapt to the interface. None of the related works present a non-invasive interface which does not require extensive structural changes. In addition, such solutions are closed systems such that if an app does not use the interface, it is excluded from the benefits of the apps using the interface.

Other literature in data sharing, such as Van Huben and Mueller (2001), propose extensive frameworks and database management systems (DBMS) to manage heterogeneous data sources, which is what is found on mobile phones between each app. In Van Huben and Mueller (2001), a DBMS for pervasive computing environments is proposed. The DBMS interfaces through a set of APIs to multiple heterogenous databases. The framework consists of query translators to map to database-specific queries. Weber *et al.* (2009) also proposes an interface for aggregation of data distributed between multiple databases in separate hospitals. The system however is designed to aggregate data and provides no mechanisms for opting into data sharing. The proposed system also requires hard coding database details into their application.

2.3 Sensor Interface

The Sensor Interface allows for data to be relayed from sensor to multiple apps through Android-specific Inter-Process Communication (IPC) mechanisms named broadcasts. The Sensor Interface also abstracts communication with sensors through a command-like interface. There are quite a few IPC mechanisms with each being generalized as low-level (shared memory, pipes, sockets) or high-level (message queues, files, memory-mapped files, Remote Procedure Call). Low-level IPC mechanisms provide great control to the developer, however also require larger understanding of Operating System details while high-level mechanisms force the user into using one mechanism and handling the execution in a particular way, such as using files, which may not be the best mechanism.

Chapter 3

METHODOLOGY

Figure 3.1 illustrates the current state of data sharing between apps versus the proposed method. In the current state, sharing data between multiple apps involves each developer developing and maintaining a custom API for accessing said app's data. Each application to support another app's data must integrate the API. The problem arises that as the number of apps to integrate with increases, each app must implement and support N-1 different APIs, with N equal to number of apps. These APIs follow no standard interface and may all be implemented in different methods, placing more burden on the developer. In the proposed framework, a Data Sharing Interface exists as a separate app which provides an unified API and acts as a registry of other apps data access information. An app still must integrate with the API, but uses the same mechanisms to register and query and the app provider is provided full control at how the query should be carried out and how much data should be returned from the query.

This work aims to be a solution by using an app which supports plug-n-play data sharing. If an app is on the system and wishes to share data, it only needs to register itself with the Data Sharing Interface of the proposed framework. Other apps as well can request shared data. If the Data Sharing Interface app is not present, no change in an app's work flow occurs. As apps begin sharing data more and integrating better, app suites may emerge. These app suites provide top-notch data sharing functionality. However, there is no fragmentation among suites because any app can always improve data integration and move between suites or perhaps exist in two suites at once.

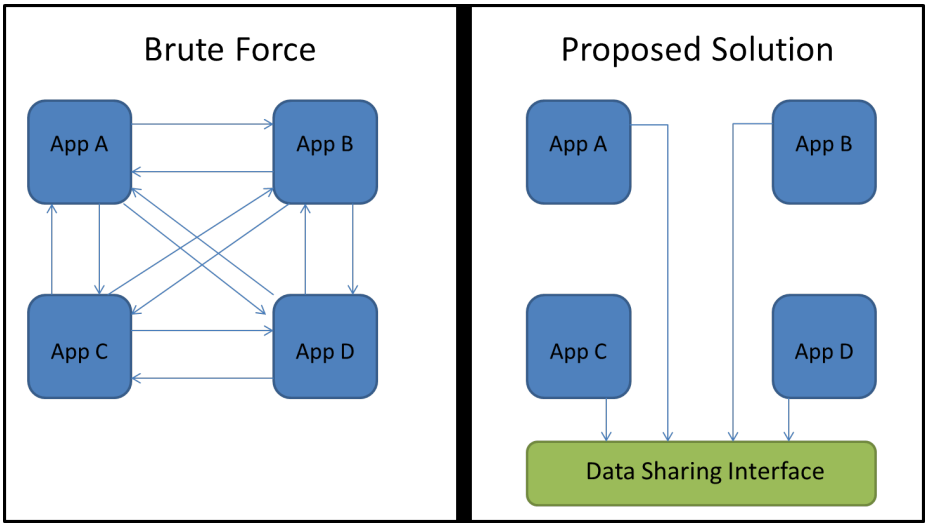

Figure 3.1: Current custom-API driven approach to data sharing versus proposed through Data Sharing Interface.

The lack of fragmentation between suites and simplicity of sharing data are the first steps to adaptive apps that are "aware" of habits and activities learned through other apps. These are the first steps for apps which optimize themselves for each user and motivate the user to use multiple apps, thus possibly improving their life further.

This work also proposes a Sensor Interface as a means to simplify development of communications between a sensor and an app. The Sensor Interface is can connect, configure and deliver data from a sensor to multiple apps on the phone. The Sensor Interface allows for sensor data to be shared among multiple apps and lowers the expertise level required to communicate with a sensor, such as an external Bluetooth-enabled sensor by abstracting away the implementation details and providing an API to control the sensor and data delivery mechanism.

In the fourth chapter I will introduce Health-Dev, a tool for developing health monitoring apps. This tool provides the structure for the data sharing functionality and generating the Sensor Interface and Data Sharing Interface code. Health-Dev will be briefly discussed with the main focus on the design and implementation of the two interfaces the proposed framework provides. The fifth chapter will describe

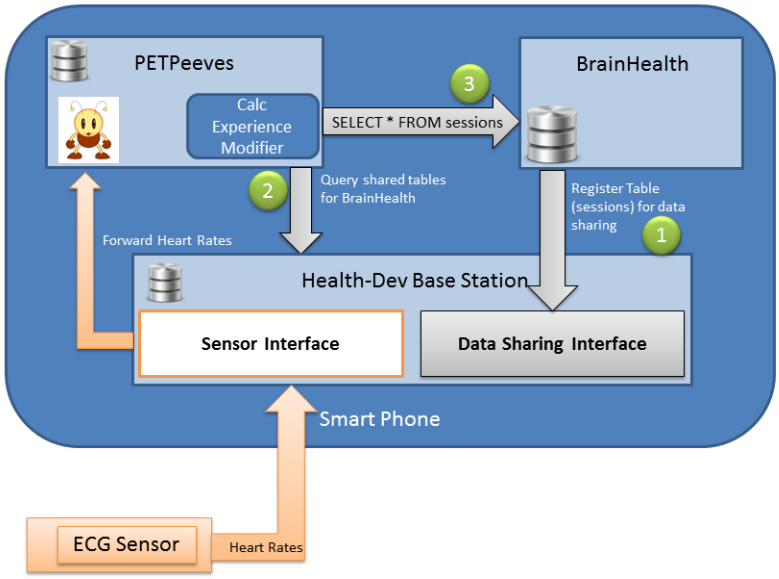

Figure 3.2: PETPeeves and BrainHealth on the smart phone; PETPeeves is aggregating ECG data through sensor interface and querying BrainHealth's shared data through data sharing interface.

the app suite and the multimodal feedback techniques used. Figure 3.2 provides an overview of how the suite interacts with each other and uses the interfaces provided to communicate with the sensor and between each other.

Chapter 4

SHARING AND CARING FRAMEWORK

As pervasive health monitoring becomes readily possible through health apps on smart phones, Pervasive Health Monitoring Systems (PHMS), which consist of a pervasive low-power sensor and a base station (phone), need to be simple to develop safe, sensor code. However, developing sensor code requires substantial domain specific knowledge including the specifics of the embedded operating system and working understanding of limited resources, deadlocks, and race conditions. In addition, the sensor code should be safe to the extent that race conditions, deadlocks, and memory management are handled correctly. For these reasons, there is a large overhead in creating a personal PHMS and thus the average user is dissuaded by the difficulty of PHMS development. In order to bring PHMS creation to the general masses, a model-based automatic code generation tool was developed to abstract the complexities of PHMSes and allow generation of sensor and smart phone code from simple models defined by the user.

4.1 Preliminaries

Health-Dev, proposed by Banerjee *et al.* (2012), is a model-based automatic code generation tool for sensors and Android phones. Health-Dev abstracts a PHMS into components with properties and connections between them. The collection of components form a PHMS model, defined in Architecture Analysis and Design Language (AADL). Health-Dev parses the model and generates sensor and smart phone code which together form a PHMS.

The PHMS model consists of Motes, Sensors, Base Stations, Communications, and Algorithms. Each of these components has properties, such as sampling frequency, communication protocols (Bluetooth, ZigBee), type of sensor (ECG, Accelerometer), and algorithms to run either sensor-side or phone-side. The smart phone code generated by Health-Dev is the Base Station of PHMS and provides out-of-the-box functionality of connecting to motes, starting and stopping sampling, changing sampling frequency, and plotting the data on a line graph. Figure 4.1 shows the different views of the Base Station app generated by Health-Dev. The views provided are optional and can be substituted by a developer. By retrofitting the views, a developer has quickly developed a health app with a custom interface and added functionality, while not having to worry about communication and parsing the sensor data.

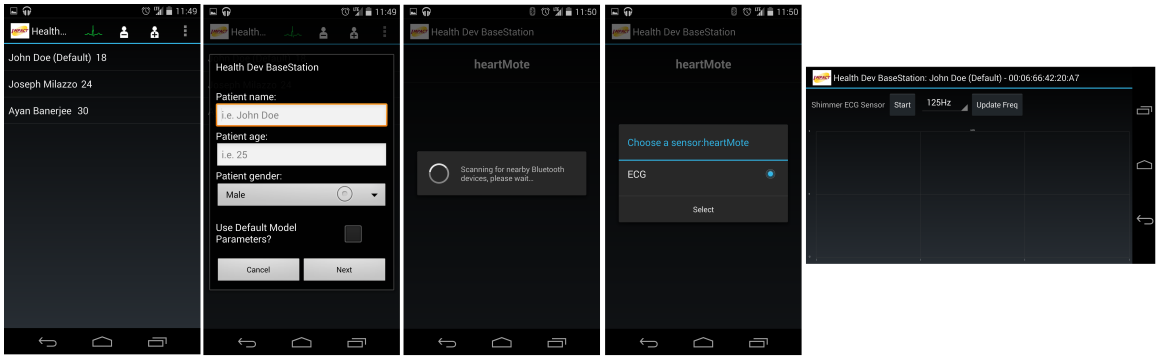

Figure 4.1: Different views of Health-Dev Base Station app.

The Base Station also provides a more prominent functionality which is brokering of communication between sensor and business logic (custom code). The Base Station exposes an Application Programming Interface (API) which allows a third-party application to register itself to receive sensor data, communicate with the sensor, add custom algorithms to run in the Base Station, and lastly register custom callbacks such that the third-party app can be made aware of certain state changes in the Base Station, such as losing connectivity with the sensor. From this point forward, the Base Station is discussed in terms of being an API.

4.1.1 Base Station

Health-Dev Base Station, also refereed to as the Sensor Interface, acts as a middleman between sensors and smart phone; communication and signal processing are handled by the Base Station; in addition, sensor data, whether processed or raw, can be forwarded to third-party apps; to provide custom business logic and integration of the data. Figure 4.2 provides an overview of how data is received and forwarded to a third-party application. The Base Station first receives a `START_SENSING` message which prompts the Base Station (already paired with sensor) to send a packet to sensor to begin sampling and sending data. As data is received, packets are parsed and raw data is passed to a pipeline of algorithms. These algorithms generally consist of different signal processing methods, but can also contain custom code registered by a third-party app. Once the algorithms are finished executing, the Base Station checks to see if any apps have registered to receive data and if so, the processed data is broadcast to the app. The app is then free to do what they like with the data. Base Station will continue this cycle until a `STOP_SENSING` message is received.

Inter-application Communication

Inter-application communication is handled through Broadcast and Broadcast Receivers in Android. Broadcast receivers are a core part of the Android OS and much of app development involves listening to various broadcasts from the Android OS to adapt to phone state, such as when the battery level updates. A Broadcast receiver is just a special type of listener for Broadcasts on the system. Broadcast receivers require a unique signature of a broadcast to be eligible to receive the broadcast. For example, the signature of a battery change event is `Intent.ACTION_BATTERY_CHANGED`. These signatures must be registered either during run-time or statically in the Android

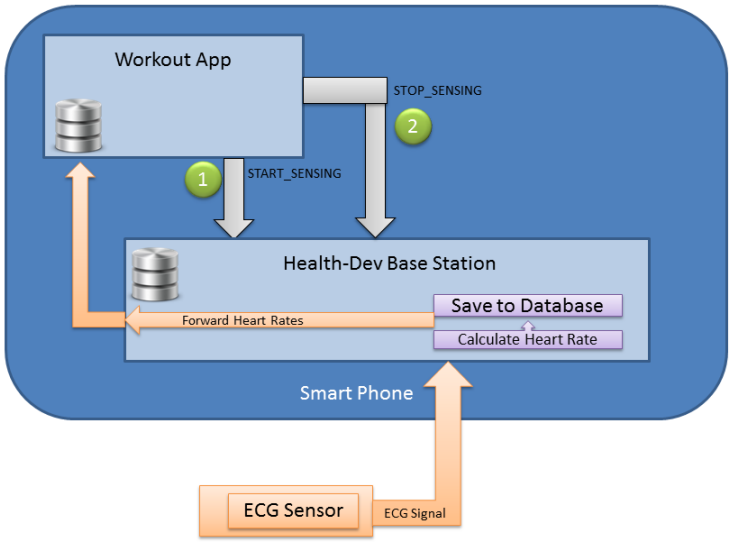

Figure 4.2: Sensor Interface receiving data from external sensor and forwarding to app.

Manifest, a document which describes the app and permissions needed.

Broadcast receivers listen for broadcasts, which are structured messages which can be sent across the OS and received by an app. There are two types of broadcasts; normal or ordered. Normal broadcasts are completely asynchronous; all receivers are run in an undefined order, often at the same time. This is more efficient, but means that receivers cannot use the results or abort the broadcast. Ordered broadcasts are delivered to one receiver at a time. Each receiver executes in turn so it can propagate a result to the next receiver; or it can completely abort the broadcast so that it won't be passed to other receivers.

The Base Station uses normal broadcast to broadcast data. Due to broadcast receivers needing to register for unique signatures of broadcasts, the Base Station provides an API for registering and reading the different signatures. Health-Dev allows for sensor communication through direct API calls or through a broadcast API.

Database

The Base Station also has the ability to store raw or processed data from the sensor in an internal database. This functionality allows for data to be buffered before broadcasting or simply to store if using the Base Station as the app itself, instead of an API. In Android, content providers manage access to a structured set of data, such as a database. They encapsulate the data and provide mechanisms for defining data security, such as which apps can access the database.

An application accesses the data from a content provider with a ContentResolver client object. This object has methods that call identically-named methods in the provider object. The ContentResolver object in the client application's process and the ContentProvider object in the application that owns the provider automatically handle inter-process communication.

In order for a client to access a ContentProvider, a content Uniform Resource Identifier (URI) is required and identifies data in a provider. Content URIs include the unique, name of the entire provider (authority) and a name that points to a table. An example of a content URI is `content://user_dictionary/words`. In this URI, the table name is "words" and "user_dictionary" is the authority. The string "content://" is always present and identifies this as a content URI.

4.2 Data Sharing Framework

App integration allows two distinct apps to be aware of another apps functionality. In the example given previously, the workout app was aware of sudden change in calorie intake and thus was able to adjust workout planning. In order for this to be achieved, partial data sharing must take place. Partial data sharing refers to exposing controlled portions of an apps' data to other apps, so as to enable integration. As

in the workout example, the workout app would need to query data of calorie intake app to be able to adjust planned workouts. In addition to partial data sharing, a method for allowing apps to opt-in to sharing data and/or receive other apps' data is necessary. I discuss these challenges below.

4.2.1 Limiting Visibility

Data sharing works by exposing portions of an app's data structure, in many cases tables in a database. For another app to query another content provider, a content URI must be known as well as the schema of that table and a description of each column. These are the core pieces of information an app must provide in order for an external query to be formed.

4.2.2 Participation

There is security concern with exposing database details in the open as well as restricting access after-the-fact. If the details were public knowledge, a change of content URI would be required as well as the complexity in changing database tables. Instead, I propose using Health-Dev Base Station as an interface to a registry, providing access to database details as well as allowing apps to enable or disable visibility of their details.

4.2.3 Implementation

The implementation of data sharing in Base Station involves using Broadcast-based API. Base Station creates a UUID (universally unique identifier) for each registered app. The app can use an optional Schema Builder class, shown in Figure 4.3, provided by Base Station API to pack the content URIs, schema, and descriptions into a custom object which is passed through a `REGISTER_DATA_SHARING` Broadcast.

Upon receiving this broadcast, Base Station generates the UUID and stores the passed information. At any time, the app can broadcast a `DATA_SHARING_CHANGE` which updates the access to the database details.

```
new BaseStationAPI.SchemaBuilder()
                    .createTable("meals")
                    .createColumn("calorie_intake")
                    .setType("integer")
                    .setRequired(true)
                    .setDefault(0)
                    .setDescription("Calorie intake for single meal")
                    .build();
```

Figure 4.3: A Builder class provided to register database details with Base Station.

A third-party app can read another app's database by first requesting a list of all app names registered. The app may then query Base Station for a specific app's database details. The query is done by using the name of the app which Base Station will translate into the unique identifier internally. Once database details are received, the third-party app can query the database as it normally would and use the results from there.

4.3 Sensor Interface

The Sensor Interface exists to simplify the communication between an app and a sensor. The interface also exists to manipulate the flow of sensor data from the sensor to multiple apps. The Sensor Interface consists of two primary threads: main thread and BluetoothService thread. The main thread handles receiving and sending of data between apps, scheduling sensor read tasks for built-in sensors, and communicating with Bluetooth thread. The BluetoothService thread handles the connection and management of said connection to a Bluetooth device.

Figure 4.4 depicts a detailed component architecture of the Sensor Interface. The Sensor Interface is responsible to maintaining information about sensors such as their

properties (i.e. Sampling Frequency) and where the sensor data is sent out. This information is first registered with the interface through an API broadcast which the BroadcastListener receives. This information is configured in the Hardware Abstraction Layer (HAL). A HAL is a software design pattern for abstracting the low-level details of a hardware. In this case, the sensor platform is abstracted to a developer by controllable properties such as sampling frequency, sending frequency and a human-friendly name. The HAL consists of mappings between different sensors platforms (i.e. TinyOS platform) and the controllable properties. For example, sampling frequency is given through Hertz. However, TinyOS does not use Hertz, but integers to represent the frequency of sampling, thus 125 Hz is mapped to 8 for TinyOS. The HAL is also responsible for scheduling tasks with Android. A task is some routine to run at a certain point in time. An example would be an alarm set for 5 a.m. in a clock app. The HAL schedules task for sampling of built-in sensors (Accelerometer) if an app requests such a sensor. The component on the main thread is the Handler; an Android mechanism for communication between threads and works very similarly to broadcasts. This handler is responsible for communication between the main thread and the BluetoothService thread. The communication consists of sending information about sensor from HAL to the actual sensor and relying received and parsed data from BluetoothService to main thread's buffers.

The BluetoothService thread consist of five components: 1) Bluetooth Service itself; 2) Handler to allow communication from BluetoothService thread to main thread; 3) PacketParser, which handles parsing and formatting packets to and from sensor; 4) BluetoothConnect thread, a short-lived thread responsible for searching and establishing connection with Bluetooth-enabled sensor; and 5) BluetoothConnection thread, the thread which maintains a connection with sensor and sends and receives data. The BluetoothService first creates the BluetoothConnect thread which performs

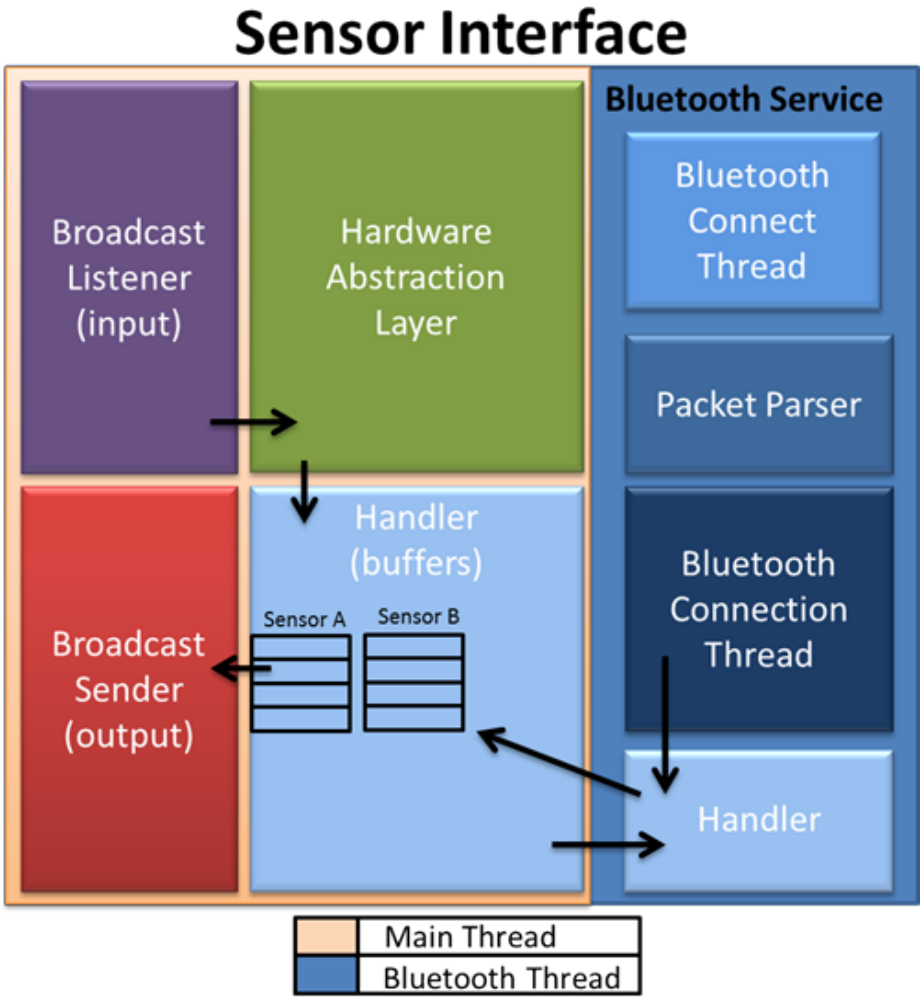

Figure 4.4: Sensor Interface architecture.

a blocking discovery operation and establishes a connection and pairing with sensor. This connection object is used in BluetoothConnection thread, which is created once connection is established. At this point, the BluetoothConnect thread is killed and the BluetoothConnection thread waits until the sensor begins sending data or a packet needs to be sent to the sensor. Data flows through the Sensor Interface starting from a sensor which sends data at a given sending frequency. The data is read by the BluetoothConnection thread and parsed into raw data by the PacketParser. The data is then sent via a Handler to the main thread which stores the data in a sensor-

specific buffer. Once the buffer is full, the data is flushed via broadcasts to apps. The data is flushed at the send rate of the sensor such that the buffer is the size of the sensor's payload. This raises the issue of perception of time by the receiving app and if the app is aware of the difference in time. From experiments performed with the Sensor Interface, the difference in time between the interface receiving data and an app receiving said data is always less than 1 second.

The Sensor Interface is capable of sending data to multiple apps simultaneously. The data delivery mechanism used are Android's broadcast mechanism to deliver the data to multiple apps asynchronously. However, in the case of two sensors (i.e. Accelerometer and ECG), the issue of in-order or out-of-order data delivery arises. Out-of-order delivery is the delivery of data packets in different order from which they were sent. This case may occur when two sensors are sampling and due to the concurrent nature, one's buffer may be flushed before the other buffer. This can be prevented by using locks to force in-order delivery, however the Sensor Interface does not enforce in-order delivery. This decision is based on the more important goal of reducing time between sensor sending data and app receiving data. By enforcing in-order delivery, in the scenario above, each buffer would incur one time slice (period of time for a thread to run in a preemptive multitasking system) by blocking so that other buffer sends first. This would also incur extra complexity and the possibility of data loss due to new data being read onto the buffer.

Chapter 5

BHEALTHY

Biofeedback is known to be an effective method to promote behavioural change. Data sharing is related to multimodal biofeedback and places a great deal of importance on the feedback signal. The feedback signal is how the user becomes aware of progress during biofeedback. With this importance, two applications were created to test the feedback signal with data sharing. These two applications reside in a suite of apps called bHealthy.

bHealthy is a suite of health monitoring apps which promotes data sharing for holistic health monitoring (Milazzo *et al.* (2013)). The suite contains two applications which interact with mental and physical health. BrainHealth is the first app in the suite and monitors mental state to foster improved concentration, increase in mood, and reduction of stress through a technique known as Neurofeedback. BrainHealth integrates with PETPeeves, an app which promotes physical exercise through a virtual pet. The integration is detailed under PETPeeves section. The apps in bHealthy all use Health-Dev Base Station for interacting with the external sensor.

5.1 Preliminaries

5.1.1 Neurofeedback

Neurofeedback is a means by which participants can learn voluntary control of their brain waves and has been applied to a range of clinical conditions such as epilepsy, attention deficit hyperactivity disorder, depression, and autism Kotchoubey *et al.* (1999), Tinius and Tinius (2000), Saxby and Peniston (1995), Egner and Gruze-

lier (2001), and Jarusiewicz (2002) as well as improve performance in healthy subjects. Neurofeedback is delivered to the patient through typically 60 minute Neurofeedback Training (NFT) sessions. These sessions can range from controlling explosions on a computer screen to controlling speed of a virtual car based on some physiological measure such as relaxation.

The Neurofeedback Training activities used in BrainHealth are based on well-known Neurofeedback protocols. Vernon *et al.* (2003) states that enhanced activity of sensorimotor rhythm (SMR) has shown increased focus, especially in patients with attention deficit disorder. SMR consist of brain waves between 12-15 Hz and are, as the name implies, linked to motor skills. NFT of SMR in patients diagnosed with attention (hyperactive) deficit disorder has show a significant increase in scores on measures of sustained attention (Shouse and Lubar (1979), Lubar and Lubar (1984), Tansey (1991), Rossiter and La Vaque (1995), Tinius and Tinius (2000)). Egner and Gruzelier (2001) found that increased SMR activity is associated with reduced commission errors and improved perceptual sensitivity on the Test Of Variables Of Attention (TOVA) as well as increases in attention related P3b event-related potential. These findings led them to conclude that SMR NFT can enhance attentional processing in healthy participants. Additionally, Raymond *et al.* (2005) shows the use of alpha/theta training to elevate mood and energy; while mock group showed a significant move towards tiredness.

5.2 BrainHealth

BrainHealth is a Neurofeedback app which aids a user to learn how to permanently overcome behavioural problems, such as lack of focus, mood depression, and high stress. BrainHealth uses Electroencephalography (EEG) to extract the user's brain waves and interprets them as positive or negative for a chosen behavioural problem

based on well-known protocols of Neurofeedback.

Neurofeedback has been found to be an effective method for encouraging healthy behavior. This app consists of three Neurofeedback Training activities: focus, mood change, and relaxation. Focus is aimed towards users who suffer from learning disabilities and need a boost in mental performance, motivation, and focus. Mood change is aimed towards users whom are not satisfied with their mood and want to achieve a more positive mood. Lastly, relaxation is aimed at any user who wants to learn how to relax in any situation.

5.2.1 System Architecture

BrainHealth uses Emotiv EEG, a commercially available EEG headset which provides 14 channels. Due to Emotiv EEG using a proprietary communication medium, direct communication between the EEG and Android phone is not feasible. Instead, the PC acts as a bridge to the phone; the EEG transmits raw data to the PC where the signal processing is performed on the data. The resulting output is transmitted to the smart phone app through Google Cloud Messaging. Google Cloud Messaging is a service in which *messages* are sent to Google's Cloud Platform. The messages are then delivered to the registered receiver via Wi-Fi or mobile networks.

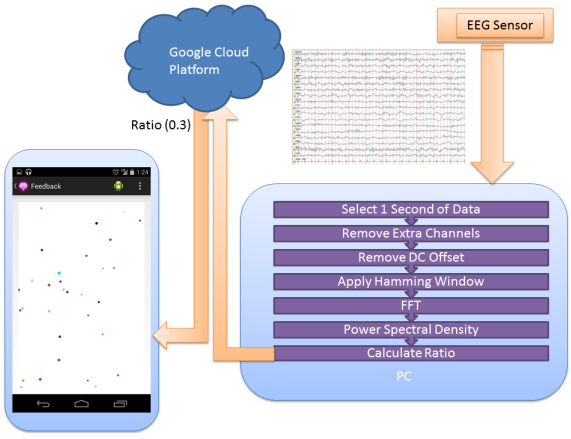

Figure 5.1: BrainHealth System Model.

5.2.2 Feedback Design

BrainHealth's feedback loop is a particle system which manipulates particles spread out on the screen. When the user is performing well, the particles are attracted towards the center and combine with each other. However, when the user's performance degrades, the particles begin to split and spread towards the edges of the screen. The color, size and positioning of the particles have no connection to the performance of the user. Figure 5.2 shows how the particle system reacts to the user's brainwaves.

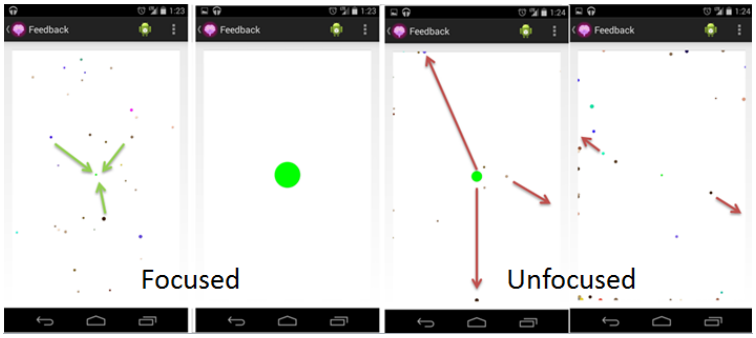

Figure 5.2: Particles moving inward due to an increase in relaxation. Particles moving outward due to decrease in relaxation.

The feedback loop's particle system is manipulated by a single ratio derived from signal processing of EEG data against NFT protocols. The calculation, seen in Figure 5.1, first filters one second of data, then removing the DC offset and channels which are not relevant for the selected NFT activity. The data is then passed through a Hamming Window and each chunk has power spectral density (PSD) estimator ran on it. A ratio based on two bands is calculated from the PSD. The bands consist of one or more ranges of frequencies the user should excite or inhibit. To calculate the ratio, the PSD of the excitement band is taken over the PSD of the inhibit band.

The average ratio is stored in a database along with the NFT activity, total time, and date. This data can be used to track progress in BrainHealth and is shared to other apps such as PETPeeves, which will be discussed in the next section.

5.3 PETPeeves

PETPeeves is an app leveraging people's bond with a virtual pet. The app aims to encourage a user to increase or sustain physical exercise through bonding and caring for the virtual pet, shown in Figure 5.3. Several surveys, such as Kanoh (2008), have shown the effectiveness of virtual pets in encouraging positive mental state in children. PETPeeves manipulates the virtual pet's mood based on physical activity of the user.

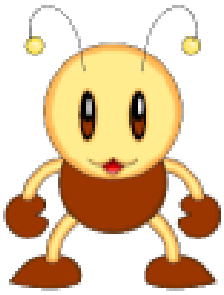

Figure 5.3: Virtual Pet used in PETPeeves.

The goal is to motivate the user to exercise and keep their pet happy or to achieve the maximum level of happiness. If the user neglects the pet's happiness, the pet will lose happiness and the user will then need to work extra hard to achieve the previous level of happiness.

PETPeeves employs a leveling system for the pet such that as the user is exercising, experience points are earned. These experience points eventually cause the pet to level up. The amount of experience points increases depending upon the pet's level. For example, 17 experience is needed for a level 8 pet, but 20 experience is needed for a

Table 5.1: Pet's experience level which maps to a specific mood.

Levels	Mood
0 - 6	Unhealthy
7 - 16	Crummy
16 - 20	Neutral
21 - 28	Pumped
29+	Ecstatic

level 16 pet. A range of levels represent a specific pet mood, which can be seen in Table 5.1.

The experience leveling formula is the same as a popular game, Minecraft created by Persson (2014). The amount of experience needed to reach the next level scales depending upon the current level. The algorithm for calculating the amount of experience until the next level is:

$$f(level) = \begin{cases} 62 + (level - 30) \text{ x } 7, & \text{if } level \geq 30 \\ 17 + (level - 15) \text{ x } 3, & \text{if } level \geq 15 \\ 17 \end{cases}$$

During exercising, experience points are added and subtracted and eventually the pet will level up or down. Since experience points are derived from physical activity, a health metric must be examined to determine if the user is exerting herself or not. The health metric used is heart rate extracted from ECG signals through an external sensor worn by the user.

5.3.1 System Architecture

Figure 5.4 depicts the application flow of the app. The user first selects a profile if already created. This switches to the exercising screen and loads the user and pet

data. This information includes user's age and weight to calculate calories burned, and pet's experience levels. At this point, experience is deducted from the pet depending on the time since last app use and a modifier is given if the user has used BrainHealth within the past week. The user then presses the start button, which broadcasts a `START_SENSING` message to Health-Dev Base Station and starts a timer. If there is no response from Health-Dev Base Station, then the user is prompted to ensure the device is properly connected to and try again. If a response is received, a new *Session* is created. A *Session* is a period of time in which the user is engaged in either PETPeeves or BrainHealth. The Session consists of the feedback metrics used in the app and elapsed time of use. In the case of PETPeeves, the Session consists of date, elapsed time, pet's experience at the start and end of the session.

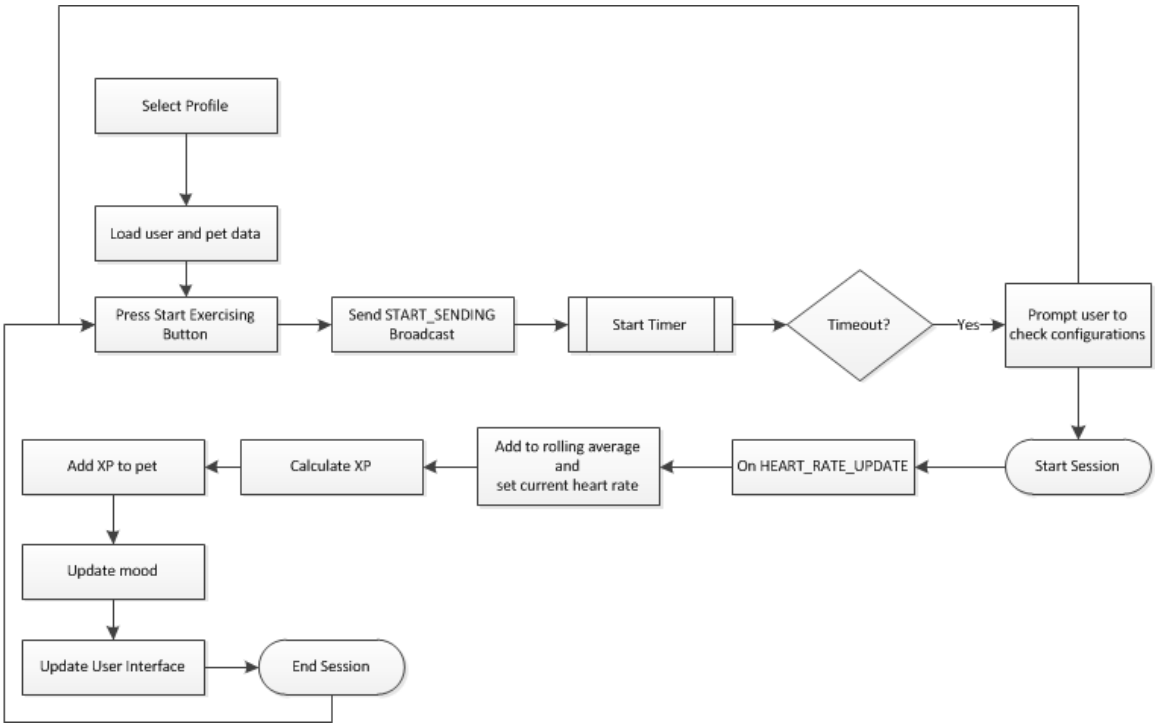

Figure 5.4: PETPeeves application flow.

Since PETPeeves has asked to start sensing data from the ECG sensor, Health-Dev Base Station is now broadcasting sensor updates to PETPeeves. In this case, the heart rate is calculated on the sensor and being forwarded to PETPeeves. On receiving of the sensor data message, the heart rate data is added to a rolling average and the current heart rate is updated. These pieces of data are then used to calculate the amount of experience to add to the pet, the mood is updated if needed, and the user interface is updated. After the user is satisfied with the progress made, the stop exercise button is pressed which stops the *Session* and broadcasts a `STOP_SENSING` message to Health-Dev Base Station to stop sensing on the sensor. The session and pet data are updated in the database and the screen resets. The user is then free to close the app or start again.

5.3.2 Calculating Experience

At the start of a session, the pet's experience is modified based on the number of days since last used. If the app has not been used for more than a day, experience is subtracted from the pet such that the experience is

$$numberOfDaysSinceLastUse experienceToNextLevel$$

.

PETPeeves calculates experience points to add/subtract on every heart rate update, which occurs once a second. During the `HEART_RATE_UPDATE` event, the new heart rate is added to a rolling average of 10 seconds as well as displaying the current heart rate to the user. The algorithm for calculating experience at each `HEART_RATE_UPDATE` event is shown below.

function $calculateXPFromECG(averageHeartRate, baselineHeartRate, moodModifier)$

$delta \leftarrow averageHeartRate - baselineHeartRate$

if $averageHeartRate \leq baselineHeartRate + 5.0$ **then**

$xp \leftarrow -1$

else if $delta \leq 10.0$ **then**

$xp \leftarrow -1$

else if $delta \leq 25.0$ **then**

$xp \leftarrow 1$

else

$xp \leftarrow 2$

end if

if $moodModifier > 0$ **then**

$bonusXP \leftarrow Math.round(Math.random() + moodModifier)$

else

$bonuxXP \leftarrow 0$

end if

return $xp + bonusXP$

end function

5.3.3 Multimodal Feedback

PETPeeves takes advantage of shared data of BrainHealth through a bonus modifier. When BrainHealth is used in conjunction with PETPeeves, a modifier is granted which gives a small chance to earn an extra experience point during the session. If the user's average ratio in BrainHealth is larger than 0.5 (performs well in NFT), then there is a 10% chance to generate an additional experience per second, else there is a 5% chance is given.

Chapter 6

EVALUATION

The proposed framework consists of 2 interface modalities: 1) Data Sharing Interface and 2) Sensor Interface. These interfaces enable data sharing between apps and data forwarding between a sensor and multiple receiver apps. This chapter focuses on evaluating the proposed framework from the perspectives of a developer and the system. From the developer's perspective, the challenge stated is to reduce the burden of integrating data sharing; which is evaluated through lines of code (LoC). From the system's perspective, the challenge stated is to accomplish the objective of this thesis without performance degradation (scalability); which is evaluated by latency of IPC.

6.1 Experiment Platform

Table 6.1: Nexus 5 Hardware Specifications.

CPU	2.3 GHz 4 Core Qualcomm Snapdragon 800 MSM8974
RAM	800 MHz 32-bit dual channel LP-DDR3 (12.8 GB/s)

The following experiments were ran on a Nexus 5 with Android 4.4 with specs seen in Table 6.1. These experiments used the app suite (PETPeeves and BrainHealth) to gather results.

6.2 Lines of Code

The measure used to represent burden on a developer was lines of code (LoC) written to incorporate a given solution for data sharing or communicating with sensor. The two comparisons made were:

Sensor Interface versus no Sensor Interface

The experiment requires connecting to an external Bluetooth-enabled ECG sensor and read raw data from it. The data does not need to be parsed and the code does not need to have any error handling, just the basics to discover, connect, and read data. Using PETPeeves as the test bed, Figure 6.1 lists the lines of code needed for the above requirements. Using the Sensor Interface greatly reduces the amount of lines of code needed to communicate with an external sensor as well as simplifies sensor communication.

Without Sensor Interface	**501**
Bluetooth Service	469
Starting and receiving data	38
With Sensor Interface	**24**
Register and send broadcasts	12
Starting and stopping sensor	2
Receiving data	10

Figure 6.1: Lines of code comparison to connect to Bluetooth-enabled sensor and read data with and without Sensor Interface in PETPeeves.

Data Sharing Interface versus no Data Sharing Interface

To not use the Data Sharing Interface would mean manually discovering, connecting, and reading of sensor data. This was already shown with not using Sensor Interface, thus it is natural to assume there is a significant difference in lines of code.

Data Sharing Interface versus Custom API

Using the Data Sharing Interface is compared to the lines of code needed using a hypothetical custom API. Table 6.2 depicts the trend for how the lines of code increases as apps increase. The formula consists of 3 variables:

Table 6.2: Comparison of lines of code between Data Sharing Framework and hypothetical custom API. A is LoC to register a table; B is LoC to perform a query; C is LoC to process the results.

# Apps	Data Sharing Interface	Custom API
1	A+B+C	B+C
2	2 x (A+B+C)	2 x (B+C)
3	3 x (A+B+C)	3 x (B+C)

- **A** is the lines of code needed to register a table with the Data Sharing Interface. This variable is specific to the Data Sharing Interface and from the experiments was found to be on average 5 LoC in PETPeeves.

- **B** is the lines of code needed to perform a query. This is common between both apps and was found to average at 7 LoC in PETPeeves.

- **C** is the lines of code needed to process the results of the query. This is a factor, but no average is given as this depends upon the specific logic of each app.

As Table 6.2 shows, using a custom API requires less lines of code than using the Data Sharing Interface. However, the results collected were for only one hypothetical custom API. If there were more APIs, then the lines of code for **B** might fluctuate, while **A** and **B** will alway be a constant number.

6.3 Scalability

To determine if the proposed framework is scalable, latency of IPC is evaluated. Latency is defined to be the difference in time between a broadcast being sent and received. In Android, broadcasts that are not serviced within 10 seconds are prompted for killing. From this, it is assumed that if latency exceeds 10 seconds, the framework is not working correctly and is at the limit for scaling. Thus the question is: How many apps can concurrently use Data Sharing Interface before becoming unresponsive?

6.3.1 Setup

To evaluate the latency of the interfaces as the number of apps increase, 3 conditions are considered as the usage models of the interfaces and served as the basis for the scalability model. These combinations are derived from different usage scenarios of interfaces and are outlined as follows:

Sensor Interface only

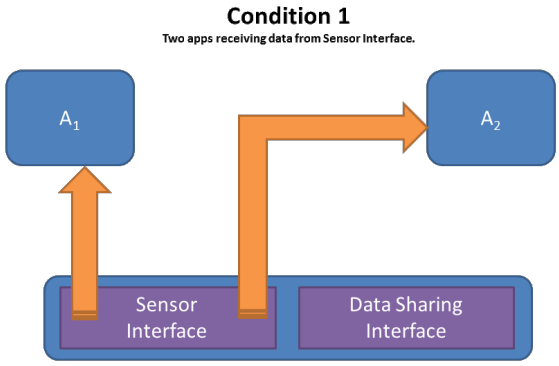

Figure 6.2: Two apps receiving data from Sensor Interface.

Condition 1 models two applications A_1 and A_2. Both applications receive X bytes of data every t seconds from the Sensor Interface. The IPC mechanism is synchronous such that a broadcast receiver order is followed and each app receives the broadcast in order from Sensor Interface.

Sensor Interface and Data Sharing Interface

Condition 2 models two applications A_1 and A_2. A_1 is receiving data through Broadcasts from Sensor Interface. A_2 is sharing data with A_1. There are 3 broadcasts and 1 Content Provider IPC call for data sharing. Like Condition 1, X bytes are received every t seconds from synchronous Sensor Interface broadcasts. The data sharing IPC

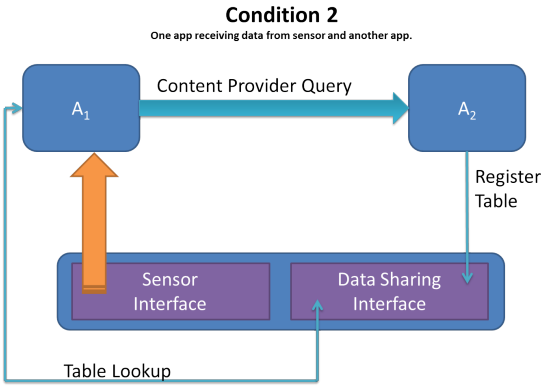

Figure 6.3: Data collection from Sensor and Shared Data Interface.

calls are asynchronous, thus Android's scheduler handles the delivery and scheduling of the calls.

Data Sharing Interface only

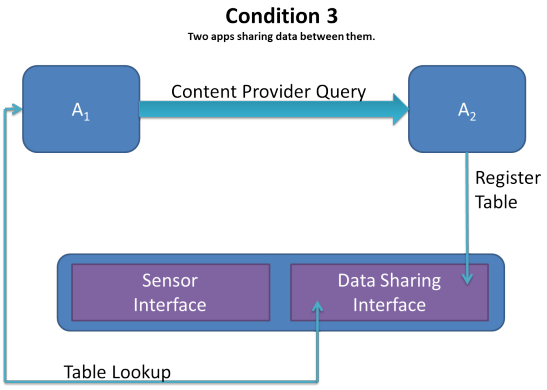

Figure 6.4: Two apps sharing data through Data Sharing Interface.

Condition 3 models apps A_1 and A_2 where A_2 is sharing data with A_1. One registration broadcast is made from A_2 and 2 broadcast and 1 Content Provider IPC call is made from A_1. This condition only considers the Data Sharing Interface.

The experiments performed are to collect latency for IPC calls under a usage model of the framework. Condition 2 was chosen as the usage model and was broken into 3 experiments. For data sharing between apps, Content Provider IPC queries are

used, while broadcasts are used for Sensor Interface data communication. For each experiment, the time for sending and receiving a broadcast is recorded.

In the experiments, payload size of a broadcast is increased from 128 bytes to 256 KB. In order to model many apps communicating over IPC, an assumption was made that if 4 apps are communicating a byte per broadcast, then then the same scenario can be modeled with 2 apps communicating 2 bytes per broadcast.

Three total experiments are conducted and outlined below:

1. A_1 and A_2 communicate through data sharing framework. Content Provider IPC queries of a fixed size are sent once a second. There is no communication with the Sensor Interface.

2. Measurement of broadcast latency over varying payload size at 1 Hz between Sensor Interface and A_1. Payload sizes are: 128 bytes, 512 bytes, 1 KB, 4 KB, 16 KB, 256 KB. No data sharing takes place.

3. Data sharing and sensor interfacing. Content Provider calls are made at a fixed rate and payload while the sensor interface broadcasts different payload sizes.

6.3.2 Results

Figure 6.5 depicts the differences in broadcast latency with and without content provider queries from the Data Sharing Interface. It can be seen that at a payload size of 4 KB, the latencies begin rising. This falls in line with Hsieh *et al.* (2013), suggesting that latency increases over broadcast IPC once the initial kernel buffer is filled. The figure shows that Data Sharing Interface IPC calls do contribute to latency of broadcasts, but the main variable is payload size. At 512 KB, the latency is greater for broadcast and content provider than with only broadcast. I believe this may be due to Garbage Collection, which introduces delays of 10-30 ms.

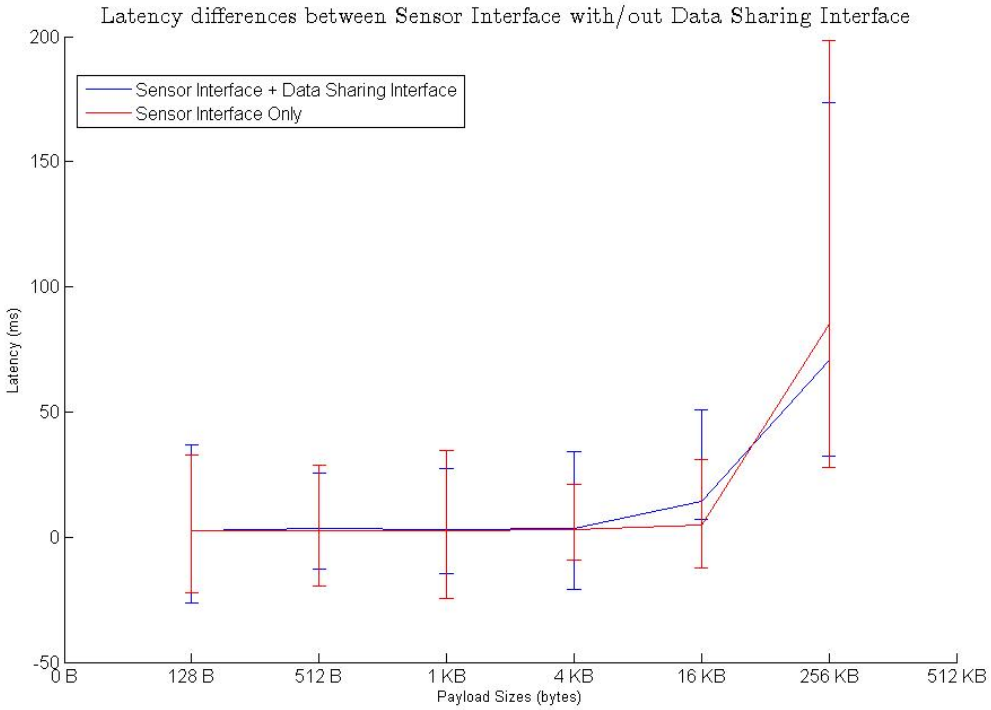

Figure 6.5: Broadcast latency differences between inter-app communication with and without Data Sharing Interface IPC.

With the highest payload and using both Sensor Interface and Data Sharing Interface at simultaneous and constant usage, even with 100 apps in this usage model, the latency of broadcast IPC would not reach the 10 second threshold imposed by Android. Since the usage model is unlikely (3-4 apps with payload size of 128 bytes is more likely), the proposed framework is assumed scalable and thus does not incur performance degradation.

Chapter 7

DISCUSSION

7.1 Limitations

In this chapter I discuss the concerns and feasibility of data sharing through my framework. The limitations I discuss are data interoperability and multi-user environments. Data interoperability is the exchange of data between two parties in which both parties understand the data. A key issue between two separate systems communicating is how the data is formatted so that both systems can understand. In many systems today, XML or JSON are popular choices. These markups allow the data to describe itself including the structure.

In my framework, data is not described in XML or JSON, however in a non-standardized way. To understand a query's data, the schema and description must first be retrieved from Data Sharing Interface and parsed. The parsing of the description is non-trivial due to having no standard and leaving the description to the registering app; thus a human reader is most likely necessary to fully understand the data format.

The reason for choosing a descriptor in the registry rather than wrapping the queried data in XML or JSON is to reduce work from the third party app, thus keeping the framework non-invasive. If the third party app were to wrap the result of a query in XML, they would be writing a basic API, which is one thing the framework aims to avoid. Instead, the user just needs to register the tables, schema, and write a short description of each field and their data is shared.

A limitation of the framework is multi-user support. An app's data is context-

sensitive to the user account; there may be multiple user accounts per phone. This creates a problem to query for a specific user's data. There are two concerns: a) user information should be private to one app and b) there may be no similar fields between the two app's implementation of a user account. An example may be if one app uses OpenID, a framework for maintaining the same login to multiple apps, while another uses OAuth, a similar framework. Two different frameworks, no similarity between their user account data; how can these two apps be aware of which user is which?

This poses a severe limit on data sharing in a multi-user environment. A possible solution may be to store a unique identifier for a user in interface and have the app implement a translation method which will be called between queries, however this adds complexity from sharing data thus violating a core principle of the framework.

Cloud storage is another limitation of the framework. Many apps have begun storing their databases in the cloud with no local database. These apps are unable to share data in the current framework. While there may be solutions such as extending the interface to query the cloud's data, these come at the price of having an app implement an API. While the framework cannot operate with cloud-based databases, the solution presented does allow for offline availability.

7.2 Multimodal Feedback

In PETPeeves, I have tried different techniques to integrate data shared from BrainHealth. These techniques were Compound Moods and Bonus Modifiers. Compound Moods was the addition of other moods based solely on the use of BrainHealth. An example would be if a user performed well in BrainHealth and was active in PETPeeves, his pet might be *Focused* and *Fit*. However no value was added to the experience of PETPeeves, just a different pet animation. Bonus Modifiers on the other

hand change the experience of PETPeeves. For using BrainHealth recently, a user is granted a small chance to gain additional experience. The better the user performs in BrainHealth, the larger the bonus experience change is. As the pet levels, the pet becomes happier thus achieving the goal of the app.

During development of PETPeeves, two versions were developed. The first version used Compound Moods while the other used Bonus Modifiers to integrate BrainHealth data into the app. For each version, the app was given to 3 lab members, the same lab members for both versions. The lab members were first told the goal and how to use the app. The lab members were then asked to use the app at least once a day for two days. After completion, the members were asked the series of questions to gauge effectiveness of the feedback. The questionnaire consisted of the following questions:

1. Do you find this app interesting?
2. While using the app, did you feel motivated to reach the goal?
3. Did you see any change in Pet's mood?
4. If you saw a change in Pet's mood, did you feel accomplished or motivated?
5. Would you use this product again?

The results from the questionnaire were aggregated and produced a steady trend that the implementation of Compound Moods did not make the user feel like they accomplished anything. However, Bonus Modifiers had good feedback especially when there was a progress bar so users could track their progress. The progress bar provided the user the ability to know how happy the pet was and what difference their exercise made. There was also clear indication that using BrainHealth provided a modifier and helped the user, while with Compound Moods there was no indication. Compound Moods may still be applicable due to the lack of polishing and instructions to the

user in the implementation used in this study. Regardless, in order for synergistic feedback to occur from data sharing, that apps need to carefully consider how to integrate data that provides something of value to the user.

Chapter 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this thesis, I develop two interfaces to simplify app development and facilitate data sharing between multiple apps on a phone. The interfaces consist of: Sensor Interface, interfacing the communication between sensor and apps; and Data Sharing Interface, to enable sharing of data between multiple apps. The first interface uses automatic code generation to provide all the code needed for a sensor, the communication with smart phone, and routing of data. This interface is shown to be useful during the development of PETPeeves app; the interface is used for routing heart rate data calculated from an ECG Sensor.

I also present an approach for non-invasive data sharing among apps to create synergistic feedback for the user. The proposed solution requires minimal changes to an app to integrate and provides data sharing access to apps on a phone-wide scale. BrainHealth, a Neurofeedback training app shares neurofeedback data that PETPeeves uses through the data sharing interface. The data is used for possible bonus experience during PETPeeves use. This provides a synergistic feedback to the user. The challenges proposed are how to enable data sharing non-invasively, give simultaneous sensor data access to multiple apps, and do so without performance degradation. The presented solution satisfies all challenges. The data sharing framework uses a separate app as a registry for shared data access. The separated app allows for a number of apps to share or use shared data without making modifications larger than a few lines of code to their app. The separated app also ensures

that as long as it is installed in the phone and at least one app has registered, then any app can query the shared data.

The second challenge is met through the Sensor Interface. The Sensor Interface allows sensor data to flow through the Sensor Interface to multiple registered apps while also abstracting the low-level details of sensor communication from the developer. The Sensor Interface was evaluated in terms of lines of code against manually connecting and reading data from a Bluetooth sensor and found that Sensor Interface needed significantly less lines of code compared to manually implementing.

The Sensor Interface and Data Sharing Interface were also evaluated in terms of latency to determine the scalability of the system as a whole. The results found that with simultaneous and constant use of both the Sensor Interface and Data Sharing Interface by 100 different apps, the latency would still be within acceptable range by Android's standards.

The interfaces proposed are non-invasive and automatically code generated. These aspects provide benefit such as faster development, reduction in human errors and effort, higher quality, and more control over how components, such as sensor and smart phone, interact and communicate with each other. The interfaces act as a separate app, which enable other apps to use shared data thus increasing the synergistic feedback.

Synergistic feedback through a collection of apps sharing data and adapting gives rise to apps that adjust and learn from other apps; new health apps which are aware of exercise or calorie intake recorded through other apps; and a system of non-fragmented apps.

8.2 Future Work

In this thesis, different techniques of integration for shared data are proposed in hopes to produce synergistic feedback and promote a user to supplement their lifestyle with other health apps. These techniques were shown to lab members to gain feedback on how motivating each was, however there is no conclusive evidence that integration of shared data produce better results than using two apps separately. To extend this work, a study should be carried out to validate the hypothesis presented in this thesis.

In addition, there are two aspects of the Data Sharing Framework that should be addressed. The first is how to maintain privacy for private app data, such as database implementation details. The second is how to manage user-specific data across multiple devices. As more apps use the cloud to backup data, user accounts are becoming more prevalent. In order for an app to share user-specific data, there needs to be some way to map users between different login systems. For example, if one app uses a web-based login and stores a unique key to identify a user, while another app uses username and password. Since there is no common data between these two apps' user data, there is no simple way to identify which user data an app needs for data sharing.

REFERENCES

Agrawal, N., A. Aranya and C. Ungureanu, "Mobile data sync in a blink", in "Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems", (USENIX, San Jose, CA, 2013), URL `https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/Agrawal`.

Banerjee, A., S. Verma, P. Bagade and S. K. Gupta, "Health-dev: Model based development pervasive health monitoring systems", in "Wearable and Implantable Body Sensor Networks (BSN), 2012 Ninth International Conference on", pp. 85–90 (IEEE, 2012).

Chun, B.-G., C. Curino, R. Sears, A. Shraer, S. Madden and R. Ramakrishnan, "Mobius: unified messaging and data serving for mobile apps", in "Proceedings of the 10th international conference on Mobile systems, applications, and services", pp. 141–154 (ACM, 2012).

Egner, T. and J. H. Gruzelier, "Learned self-regulation of eeg frequency components affects attention and event-related brain potentials in humans", Neuroreport **12**, 18, 4155–4159 (2001).

Gu, T., H. K. Pung and D. Q. Zhang, "A middleware for building context-aware mobile services", in "Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th", vol. 5, pp. 2656–2660 (IEEE, 2004).

Gupta, S. K., T. Mukherjee and K. K. Venkatasubramanian, *Body Area Networks: Safety, Security, and Sustainability* (Cambridge University Press, 2013).

Hsieh, C.-K., H. Falaki, N. Ramanathan, H. Tangmunarunkit and D. Estrin, "Performance evaluation of android ipc for continuous sensing applications", ACM SIGMOBILE Mobile Computing and Communications Review **16**, 4, 6–7 (2013).

Jarusiewicz, B., "Efficacy of neurofeedback for children in the autistic spectrum: A pilot study", Journal of Neurotherapy **6**, 4, 39–49 (2002).

Kanoh, H., "Education for the net generation", URL `http://www.childresearch.net/papers/digital/` (2008).

Kotchoubey, B., U. Strehl, S. Holzapfel, V. Blankenhorn, W. Fröscher and N. Birbaumer, "Negative potential shifts and the prediction of the outcome of neurofeedback therapy in epilepsy", Clinical Neurophysiology **110**, 4, 683–686 (1999).

Lubar, J. and J. Lubar, "Electroencephalographic biofeedback of smr and beta for treatment of attention deficit disorders in a clinical setting", Biofeedback and Self-regulation **9**, 1, 1–23 (1984).

Milazzo, J., P. Bagade, A. Banerjee and S. K. Gupta, "bhealthy: a physiological feedback-based mobile wellness application suite", in "Proceedings of the 4th Conference on Wireless Health", p. 14 (ACM, 2013).

Persson, M., "Minecraft", URL https://minecraft.net/ (2014).

Raymond, J., C. Varney, L. A. Parkinson and J. H. Gruzelier, "The effects of alpha/theta neurofeedback on personality and mood", Cognitive Brain Research **23**, 2, 287 – 292, URL http://www.sciencedirect.com/science/article/pii/S0926641004003155 (2005).

Rossiter, D. T. R. and T. J. La Vaque, "A comparison of eeg biofeedback and psychostimulants in treating attention deficit/hyperactivity disorders", Journal of Neurotherapy **1**, 1, 48–59 (1995).

Saxby, E. and E. G. Peniston, "Alpha-theta brainwave neurofeedback training: An effective treatment for male and female alcoholics with depressive symptoms", Journal of clinical psychology **51**, 5 (1995).

Shouse, M. N. and J. F. Lubar, "Operant conditioning of eeg rhythms and ritalin in the treatment of hyperkinesis", Biofeedback and Self-regulation **4**, 4, 299–312 (1979).

Tansey, M. A., "Wechsler (wisc-r) changes following treatment of learning disabilities via eeg biofeedback raining in a private practice setting", Australian Journal of Psychology **43**, 3, 147–153 (1991).

Tinius, T. P. and K. A. Tinius, "Changes after eeg biofeedback and cognitive retraining in adults with mild traumatic brain injury and attention deficit hyperactivity disorder", Journal of Neurotherapy **4**, 2, 27–44 (2000).

Van Huben, G. A. and J. L. Mueller, "Methods for shared data management in a pervasive computing environment", US Patent 6,327,594 (2001).

Vernon, D., T. Egner, N. Cooper, T. Compton, C. Neilands, A. Sheri and J. Gruzelier, "The effect of training distinct neurofeedback protocols on aspects of cognitive performance", International Journal of Psychophysiology **47**, 1, 75 – 85, URL http://www.sciencedirect.com/science/article/pii/S0167876002000910 (2003).

Vivo, A. C., "Inter-app communication library", URL https://github.com/tapsandswipes/InterAppCommunication (2013).

Weber, G. M., S. N. Murphy, A. J. McMurry, D. MacFadden, D. J. Nigrin, S. Churchill and I. S. Kohane, "The shared health research information network (shrine): a prototype federated query tool for clinical data repositories", Journal of the American Medical Informatics Association **16**, 5, 624–630 (2009).