SHARING IS CARING: A DATA EXCHANGE FRAMEWORK FOR CO-LOCATED MOBILE APPS

Joseph Milazzo, Priyanka Bagade, Ayan Banerjee, and Sandeep K.S. Gupta

IMPACT LAB (https://impact.asu.edu/),School of Computing Information Decision Systems Engineering, Arizona State University, Tempe, Arizona
{joseph.milazzo,pbagade,abanerj3,sandeep.gupta}@asu.edu

ABSTRACT

Data sharing between mobile apps that are co-located in a smartphone may lead to synergistic benefits to the user especially in the health-care domain. Mobile apps that monitor and control user behavior can interact to engage the user into a healthy schedule over a long term. In the current app development paradigm, apps are being developed individually and agnostic of each other. To enable interaction between apps in secure manner, collaboration between developers is needed, which can be problematic on many levels. Current approaches to app integration require large code modifications to reap the benefits of shared data such as requiring developers to provide APIs or ensuring security through elaborate measures. In order to promote app interaction, this paper proposes a non-invasive secure interface for data sharing between mobile apps. A separate app, called the Health-Dev data manager, acts as a registry to allow apps to register database tables to be shared. Two health monitoring apps are developed to evaluate the sharing framework and different methods of data integration between apps. The health monitoring apps have shown non-invasive solutions can provide data sharing functionality without large code modifications and collaboration between developers.

KEYWORDS

Health apps, Data sharing, Mobile computing, Smart phone, Application Programming Interface, Data security, Data Management.

1. INTRODUCTION

Mobile computing [12] using mobile apps is becoming very prevalent. Mobile apps are being developed which are increasingly smart which use of context-sensing technologies. A user typically uses several related mobile apps on her smart phone. Data sharing between some of these multiple collocated mobile apps may make them more context aware with a relatively lower resource requirements. For example, let's consider different applications a) PETPeeves [1], which monitors the exercise patterns using accelerometer and Global Positioning System (GPS) data, and b) continuous electrocardiogram (ECG) monitoring application. If PETPeeves has access to

Sundarapandian et al. (Eds) : CCSEIT, DMDB, ICBB, MoWiN, AIAP - 2014
pp. 61–82, 2014. © CS & IT-CSCP 2014 DOI : 10.5121/csit.2014.4808

heart rate from the ECG app, it can compute calories burnt during exercise without interfacing with a heart sensor. In general, app integration has been found to be beneficial in increasing usage through methods such as experience modifiers and unlocking difficulty levels.

Unfortunately, in the current state, data sharing is not easily achievable due to data privacy issues [2]. If an app wishes to read data from another app, either one of two approaches is taken: the developers contact each other and share internal details of data storage including data read and write permissions; 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. The API does not have the capability to restrict access to the data and hence may result in data leakage to unauthorized apps. A solution is to write custom APIs for each app attempting to access data, however, this places a burden on the developer to develop and maintain potentially large number of APIs.

In this paper, we propose a framework that enables app integration in a manner secure as well as non-invasive to the developer. The core of this framework is the Health-Dev [7] data manager app (HDDM) that maintains a registry of all available app databases. HDDM is considered as a trusted entity and any registered app opens up database access to the HDDM. Thus, instead of writing a custom API, the developer only needs to register tables of their database with the HDDM before another app can query details from the framework. Figures 1 and 2 illustrates how the framework improves data sharing. Without the framework (Figure 1), a custom API is developed for ECG monitoring app and thus PETPeeves can use the API to access the database. However, with the framework (Figure 2), ECG monitoring app first registers a table with the HDDM; PETPeeves can then query the HDDM for ECG monitoring app's database.

One interesting difference between two systems shown in Figures 1 and 2 is with the proposed framework external sensor communicates with the HDDM app rather than the ECG monitoring app. A benefit of this is that the sensor data may be shared among multiple apps in real-time. Thus, in addition to data sharing, the framework can interface sensors with apps. With the data manager handling the communication between the sensors and the apps, the possibility of another app making use of the same sensors' data exists and leads to shared functionality which is otherwise difficult to implement. This will be discussed further in Section III-A.

The proposed inter-app data sharing framework is non-invasive. 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 user's activities and physiological data through shared data from other apps in only a few steps. To validate the framework, this paper uses a suite of apps, bHealthy [1], designed to share data to provide synergistic feedback and motivate the usage of multiple behavioral health monitoring apps.

An added advantage of this data sharing framework is that it can also be used as an interface to external sensors. The HDDM can communicate with multiple sensors, create databases sensed signals and provide simultaneous access to multiple apps.

Challenges: The principal challenges in data sharing between mobile apps are as follows:

1) Trade off between security and programming workload: In the current app development methodology, to share data in a privacy assured manner, app developers need to manually provide database access to individual apps. To reduce the burden of programming, app developers might make app databases globally accessible. However

that may give rise to serious security vulnerabilities [13]. The research question that we consider in this paper is, how to enable secure data sharing between apps without developer intervention?

2) synergistic feedback: When shared data is included into an app, the integration of data is vital to providing feedback.
How can shared data be integrated to produce synergistic feedback?

Contributions: Main contributions of the paper are summarized below:

1) Health-Dev data manager interface for secure data sharing between apps;
2) The HDDM service to interface sensor and app which can route sensor data to multiple apps.
3) Demonstration of synergistic benefits obtained due to secure sharing of data between apps;

2. EXISTING TECHNIQUES FOR APP INTEGRATION

This section 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 section first reviews the current state of inter-app communication across iOS and Android, the two most popular mobile operating systems. This section discusses the perceived limitations of the current state and compare related works against different granularity of changes needed.

A. Android

In Android data sharing between applications can happen if the data is stored in a structured format such as a data base table. As shown in Figure 3, to enable data access by any other app, the App 1 has to define a data base schema. This schema has to be agreed upon by the App 2 in order for it to read data from App 1. Hence at this step it requires collaboration between app developers to decide and design a common schema. On top of this schema App 1 has to define a content provider that provides access to the underlying data base. In recent Android OS every content provider is restricted to the App from which it is declared. In order to provide access to the content provider to App 2 the developer of App 1 has to set the access permissions. App 1 developer can either open up its content provider to all apps, which is often undesirable given the private nature of data, such as health data, or the App 1 can give access to only App 2. In such a case App 1 and App 2 developers need to collaborate and decide on a security mechanism. Based on the data base schema and the content provider security mechanisms the App 2 then implements a content resolver through which it can access data bases from the content provider of App 1. Once a content resolver is written in App 2, App 1 has to implement a broadcast messaging service to send notifications to App 2 whenever new data is ready. App 2 has to implement a broadcast receiver that will invoke the content resolver whenever new data from App 1 is available.

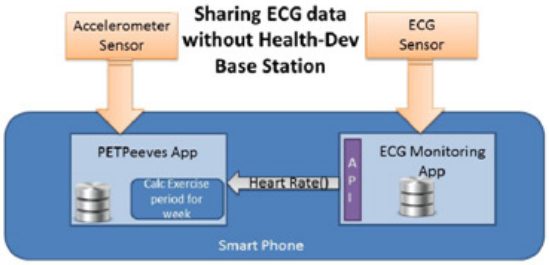

Fig. 1. System model of sharing meals table between apps without framework.

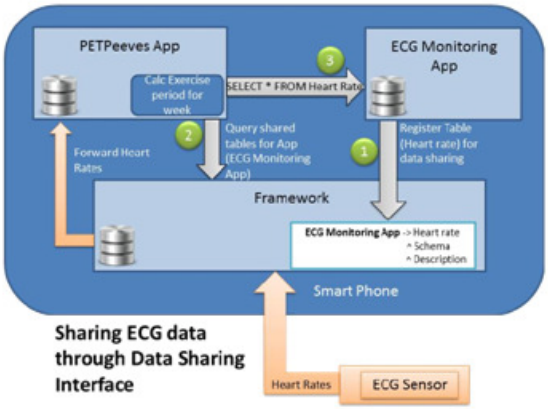

Fig. 2. System model of sharing meals table between apps with framework.

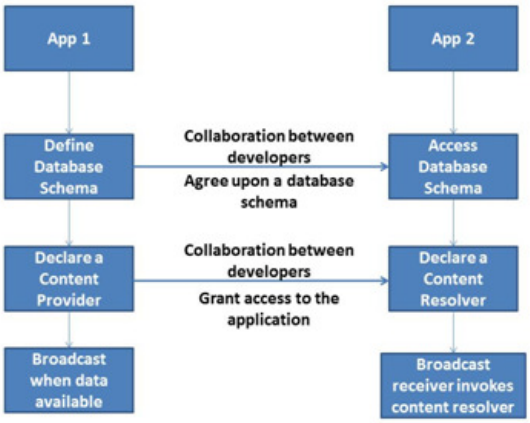

Fig. 3. Inter App interaction for Android platforms.

B. iOS

iOS runs apps in sandboxes which provide limited means of communications 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. URL schemes are currently used by many apps to launch other apps and pass basic data through URL parameters,

but 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.

The authors in [3] propose an Open Source library, Inter-App Communication, to expand 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 play an important functionality for data sharing as they allow an app to develop an API for data sharing that can return results to the requesting app.

Table Comparison Of Different Interfaces

I. 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
Health-Dev data man-Ager	Optional	None	Yes	Yes

C. Existing Works

Mobius is a middleware for interfacing complex data management schemes [4]. It provides programming abstractions of logical tables of data that spans devices and clouds. Applications using Mobius can asynchronously 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 called Simba simplifies the complexities of synchronizing data with the cloud; hence reducing the developer's work [5]. The Service-Oriented Context-Aware Middleware (SOCAM) enables rapid prototyping of context-aware services in pervasive computing environments [6]. SOCAM provides a middleware of components to define context providers, interpreters, and the interaction between different components. Their middleware can allow data to be interpreted different between multiple apps; however each app must use the middleware to participate.

Table I 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.

3. HEALTH-DEV DATA SHARING FRAMEWORK

The Health-Dev data manager is a background service running in a smartphone. It maintains two registries: a) registry for App databases, which maintains a log of the data base names and schemas that an App has made available for access, and b) a registry of permissions set for each shared data base. The methodology by which two apps can share data through Health-Dev is shown in Figure 4.

Two apps that want to communicate with each other should first register with the Health-Dev Data Manager service. The registration process ensures that the App and the Health-Dev service agrees upon a security protocol and a secure key. On creation of each database the App 1 broadcasts the name of the database and its schema to the Health-Dev service. The Health-Dev service stores the database schema and maintains the registry of databases. App 1 then implements content providers for each data base and uses the security primitives agreed upon with the Health-Dev to securely update the database in the Health-Dev data manager. Database updates are done using broadcasts between App 1 and the Health-Dev data manager.

Another App 2 which participates in the data sharing methodology also registers with Health-Dev and establishes a secure channel. App 2 can query the registry of Health-Dev and get a synopsis of the data bases that are available for access. App 2 can select the database that it wants to access and queries the Health-Dev registry to obtain the data base schema. The App 2 implements content resolvers for each database it wants to access. The App 2 then implements a broadcast receiver for each database and listens to database update broadcasts from the Health-Dev Data Manager.

A. Health-Dev Data Manager

The Health-Dev data manager is a background service in the smartphone. The primary functionality of Health-Dev data manager is to maintain a repository of all share-able database tables related to a registered app. Further, the data manager also handles the communication between sensors and application code, which is implemented using Health-Dev an automated code generator [7]. It 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 data manager, and lastly register custom callbacks such that the third-party app can be made aware of certain state changes in the data manager, such as loosing connectivity with the sensor.

Health-Dev data manager acts as a middleman between sensors and the smart phone; handling communication and signal processing is handled by the data manager; 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 5 provides an overview of how data is received and forwarded to a third-party application. The data manager first receives a START_SENSING message which prompts the data manager (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 data manager 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 use the data in whatever manner it wants. The data manager will continue this cycle until a STOP_SENSING message is received.

B. 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 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 data manager uses normal broadcast to broadcast data. Due to broadcast receivers needing to register for unique signatures of broadcasts, the data manager 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.

C. Database

The data manager 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 data manager 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.

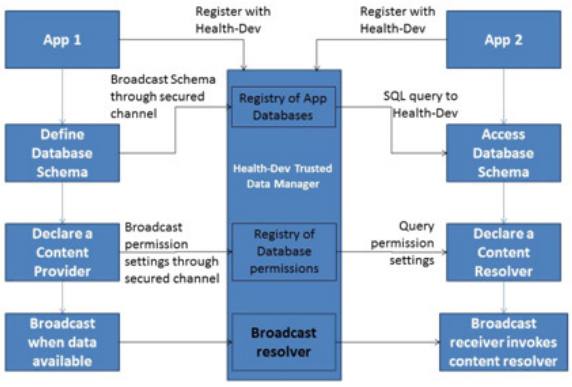

Fig. 4. Data sharing methodology for two apps using Health-Dev Data Manager.

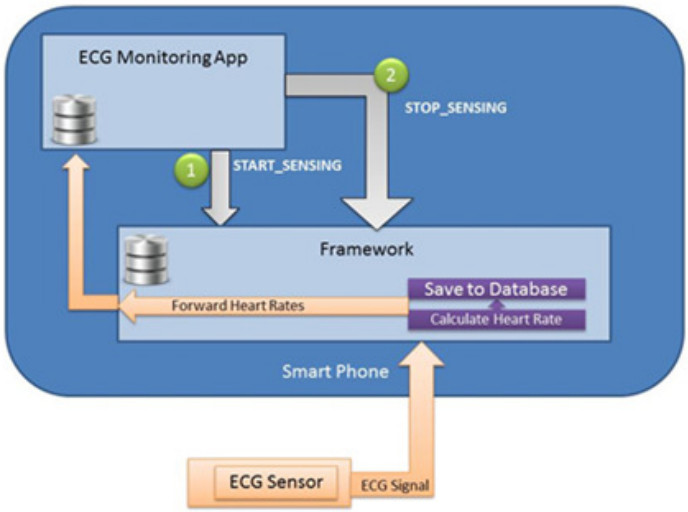

Fig. 5. Health-Dev data manager receiving data from external sensor and forwarding to app.

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.

D. 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.

E. 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 would increase. Instead, we propose using Health-Dev data manager as an interface to a registry, providing access to database details as well as allowing apps to enable or disable visibility of their details.

F. Implementation

The implementation of data sharing in data manager involves using Broadcast-based API. data manager creates a UUID (universally unique identifier) for each registered app. The app can use

an optional Schema Builder class, shown in Figure 6, provided by data manager 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, data manager 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();
```

Fig. 6. A Builder class provided to register database details with data manager.

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 data manager for a specific app's database details. The query is done by using the name of the app which data manager 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. EXAMPLE APPLICATION SUITE: BHEALTHY

Healthy [1] is a suite of health monitoring apps which promotes data sharing for holistic health monitoring. 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 data manager for interacting with the external sensor.

A. BrainHealth

BrainHealth is a neurofeedback app which aids an 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 behavioral 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.

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.

Fig. 7. BrainHealth System Model.

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. Figure 8 shows how the particle system reacts to the user's brainwaves.

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 7, 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 Section IV-B.

B. 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 9. Several surveys, such as [8], have shown

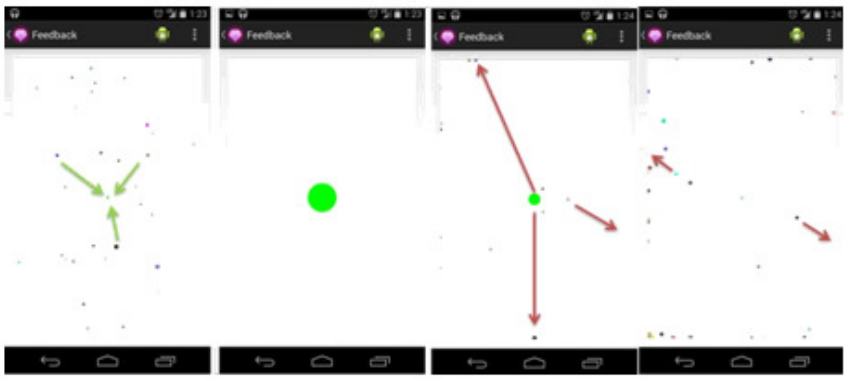

Fig. 8. Particles moving inward due to an increase in relaxation. Particles moving outward due to decrease in relaxation.

Table Ii.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

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.

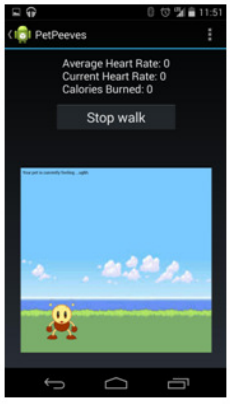

Fig. 9. Virutal Pet used in PETPeeves.

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 becomes increasingly unhappy 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 level 16 pet. A range of levels represent a specific pet mood, which can be seen in Table II.

The experience leveling formula is the same as a popular game, Minecraft created by [9]. The algorithm for calculating the amount of experience until the next level is:

During exercising, experience points are added and subtracted and eventually the pet will level up or down. Since experience

```
function experienceForNextLevel(currentLevel)
    if currentLevel ≥30 then
      return 62 + (currentLevel -30)* 7
    else if currentLevel ≥ 15) then
    return 17 + (currentLevel -15) *3

    else
    return17
    end if
    end function
```

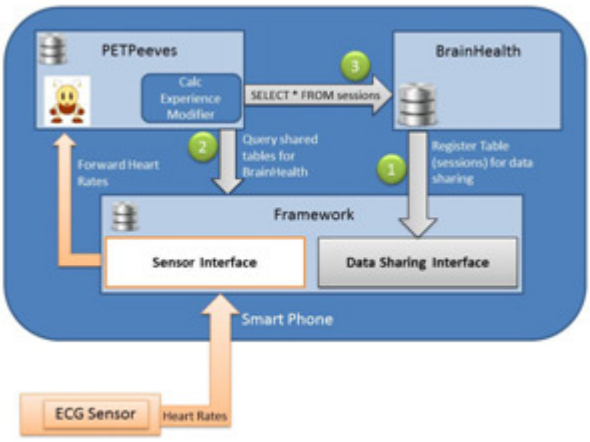

Fig. 10. PETPeeves and BrainHealth on the smart phone; PETPeeves is aggregating ECG data through sensor interface and querying BrainHealth's shared data through data sharing interface.

points are derived from physical activity, a health metric must be examined to determine if the user is exerting himself or not. The health metric used is heart rate extracted from ECG signals through an external sensor worn by the user.

5. APP INTEGRATION IMPLEMENTATION USING HEALTH-DEV

The HDDM is used to share data between PETPeeves and BrainHealth as shown in Figure 10. It shows the data sharing interface as well as the sensor interface. BrainHealth shares mental state data to PETPeeves. A positive mental health such as high focus level helps the user to get experience points in PETPeeves. Apart from the data sharing framework, the PETPeeves app uses the sensor interface to access the ECG sensor data.

Figure 11 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 data manager and starts a timer. If there is no response from Health-Dev data manager, 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.

Since PETPeeves has asked to start sensing data from the ECG sensor, Health-Dev data manager 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 data manager 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.

A. 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)

delta ← averageHeartRate – baselineHeartRate

```
    if averageHeartRate ≤baselineHeartRate + 5.0
then
        xp ← -1

    else if delta ≤10.0 then xp ← -1

    else if delta ≤25.0 then xp ←1
    else

    xp ← 2 end if
    if moodModifier > 0 then

        bonusXP ← Math.round(Math.random() + moodModifier)

    else  bonuxXP ←0 end if

    return xp + bonusXP end function
```

B. Synergistic 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.

6. SYSTEM VERIFICATION

Evaluation: The proposed framework consists of two interface modalities: 1) Sharing Data Interface and 2) Sensor Interface. These interfaces enable data sharing between apps and data forwarding between a sensor and multiple receiver apps. These interfaces heavily rely on the underlying Inter-Process Communication (IPC) mechanisms provided by Android. With the proposed framework with app suite, the data transmission latency is within accepted limit. This section focuses on, given a proposed framework, how latency scales with increased number of apps.

A. Setup

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

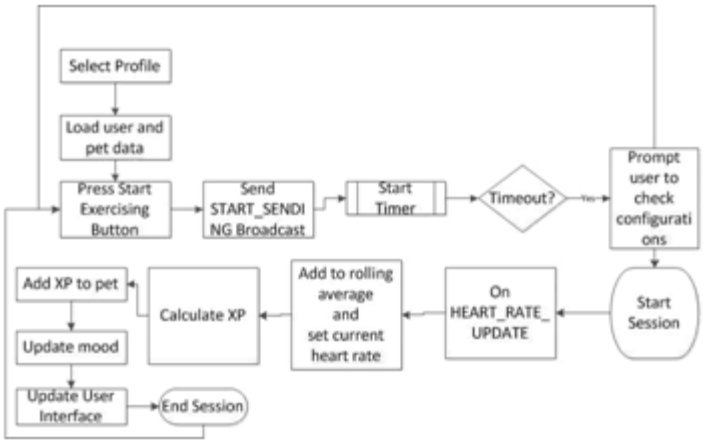

Fig. 11. PETPeeves application flow.

1) scenario 1: scenario 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.

2) scenario 2: scenario 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 scenario 1, X bytes are received every t seconds from synchronous Sensor Interface broadcasts. The data sharing IPC calls are asynchronous, thus Android's scheduler handles the delivery and scheduling of the calls.

3) scenario 3: scenario 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 scenario only considers the Data Sharing Interface.

B. Model

Latency is dependent upon the scheduling of tasks (IPC calls) by the underlying scheduler of Android. Android is based on Linux 2.6 kernel and uses Complete Fairness Scheduler (CFS), which is an implementation of a well-studied algorithm, weighted fair queuing (WFQ). WFQ is a data packet scheduling technique allowing different scheduling priorities to statistically multiplex data flows. Each data flow is represented by a separate FIFO queue and an average data rate is calculated according to the equation below. In the equation, R is the total bandwidth in the system and w_i is the size of each queue.

$$Average\_Data\_Rate = \frac{\{R \times w_i\}}{\sum_i w_i} \qquad (1)$$

Using this equation for determining average bandwidth needed to keep all queues as balanced as possible, it can be applied to servicing a process' work. If each process has a queue with weight w_i, where w_i is the total size of work in queue (size of queue). The scheduler must balance the queues each time slice so that no process is neglected. We assume that Android's scheduler (CFS) operates in process detailed above.

The authors in [10] benchmark the IPC mechanisms provided by Android. In their results, latency increases significantly for IPC calls with a payload size over 4 KB. They note this is due to the initial kernel buffer being 4 KB and thus an allocation of temporary kernel buffer is required to transfer the payload. Bearing payload size in mind and considering the working of CFS, we have defined the latency model as:

$$L = \frac{\sum_{i=1}^{N} payload\_size_i + overhead}{R} \tag{2}$$

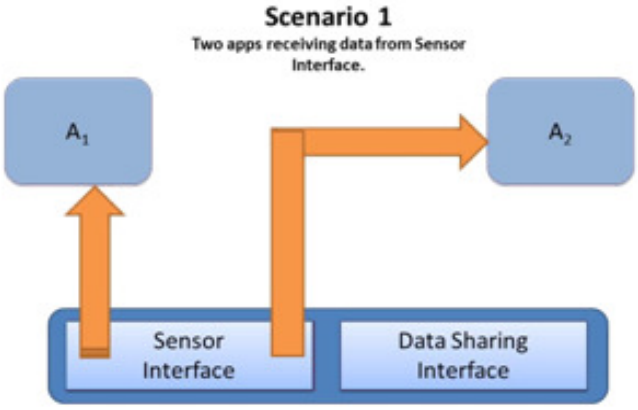

Fig. 12. Two apps receiving data from Sensor Interface.

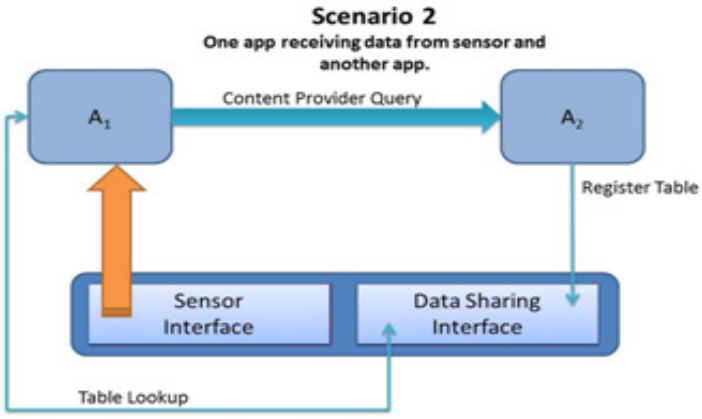

Fig. 13. One app receiving data from Sensor Interface and another app through Shared Data Interface.

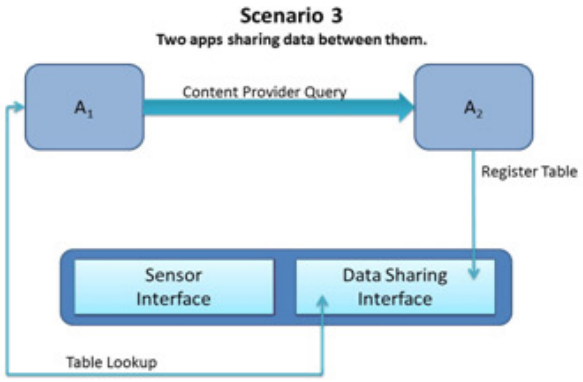

Fig. 14. Two apps sharing data through Shared Data Interface.

The total latency is the sum of all app's payload sizes (size of each queue) plus some overhead from the system divided by the total bandwidth the system provides. Overhead in smart phone can be created from events such as user interaction, incoming SMS, network connectivity, or phone calls. In Android, broadcasts that are not serviced within 1 second are prompted for killing. From this, it is assumed that if latency exceeds 10 second, the framework is not working correctly and is at the limit for scaling. Thus, L should be less than or equal to 10, L <= 10.

To validate the model, experiments were performed to collect latency data and test correctness of model proposed.

C. Experimental Setup

The experiments performed are to collect latency for IPC calls under a usage model of the framework. scenario 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. Broadcasts are sent once per second and payload sizes varies from 128 bytes to 256 KB.

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

The experiments were performed on a Nexus 5 smart phone, the hardware specifications are shown in Table III.

TABLE III. 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)

D. Results

Figure 15 depicts the differences in broadcast latency with and without content provider queries. It can be seen that at a payload size of 4 KB, the latencies begin rising. This falls in line with [10], suggesting that latency increases over broadcast IPC once the initial kernel buffer is filled. The figure shows that content provider IPC calls does play a difference in 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. We believe this may be due to Garbage Collection, which introduces delays of 10-30 ms.

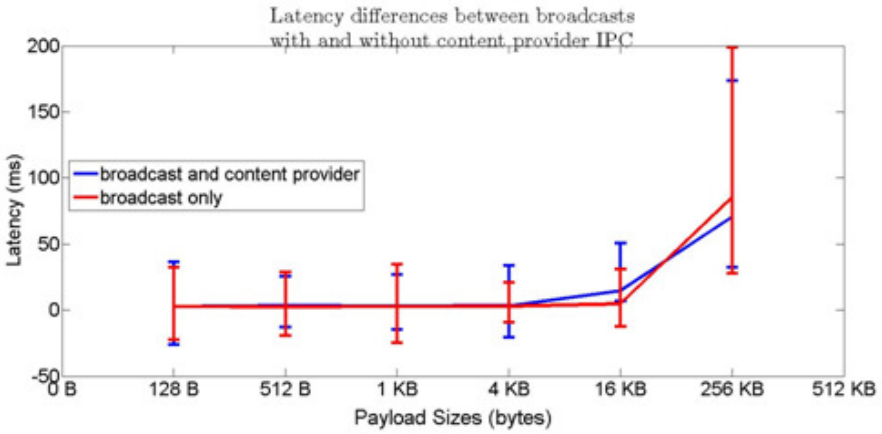

Fig. 15. Broadcast latency differences between inter-app communication with and without content provider IPC.

Content Provider IPC latency is compared with effect of broadcast and without in Figure 16. The latency without broadcasts produced from Data Sharing averages at 1 ms while without, the latency averages at 0.65 ms. Content providers are much faster than broadcasts, even with larger payload sizes (9 MB). However, the latency of a content provider is greatly affected by the payload of broadcast communication in the app. As the broadcast payload size increases, the latency also increases.

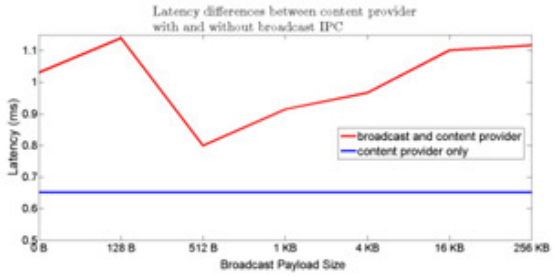

Fig. 16. Content Provider latency differences between inter-app communication with and without broadcast IPC.

7. DISCUSSIONS

A. Limitations

In this section, we discuss the concerns and feasibility of data sharing through the proposed framework. The limitations we 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 the proposed 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.

B. Synergistic feedback

In PETPeeves, we 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 used Compound Moods while the other used Bonus Modifiers to integrate BrainHealth data into the app. we deployed the app on lab members and asked for feedback on the app as to whether they felt motivated while using to use BrainHealth. Compound Moods had negative reactions while we believe due to providing nothing for the user in return for using BrainHealth. However, Bonus Modifiers had good feedback especially when there was a progress bar so users could track their progress. We believe in order for synergistic feedback to occur from data sharing, that apps need to carefully consider how another app's data can provide something valuable to the user.

8. CONCLUSIONS

In this paper, we presented a secure non-invasive data sharing framework among apps to provide 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 faced were how to enable data sharing non-invasively and how to share only what is necessitated. The presented solution resolves both 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 providing registration of specific tables to be shared, rather than whole databases. By also providing descriptions of each column in a table, some fields may be undocumented so as to protect possible information. While this challenge is met, there are possible complications that should be improved. This solution does not fully restrict access to

other tables of databases by not performing SELECT queries for the requesting app. By allowing the requesting app to make queries themselves, through carefully constructed queries, the app may gain knowledge of other tables in database.

The last challenge faced was how shared data can be integrated to produce synergistic feedback. This challenge is the combining of two apps data to enhance functionality or promote using two apps to supplement a user's lifestyle needs rather than one. This challenge is explored through bHealthy [1], the suite of health apps utilizing shared data from Neurofeedback to enhance physical exercise app. Compound moods and bonus modifiers were explored where bonus modifiers showed a positive response from lab members.

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.

An important component of any smartphone application is uploading data to a cloud server. In principle, data sharing between a smartphone application and a cloud service is similar to the problem of inter app data exchange. Although we have not explored solutions to the cloud integration problem in this work, an initial educated guess suggests that a solution through the usage of HDDM can be proposed. For example, the HDDM can allow registration of cloud services. In such a case, a cloud service can query the HDDM service in the phone regarding the availability of databases from apps. The secure communication of data between the HDDM and cloud service can be carried out using commonly used techniques such as https or digital signatures. In one of our previous work, we have proposed a solution that not only provides the cloud services access to databases obtained from physiological monitoring applications in the smartphone, but also encrypts the communication using the physiological signals. The physiological value based end to end security protocol (PEES) [11] shows the feasibility of non-invasive cloud integration through a trusted data manager such as HDDM.

Future Work: In this paper, 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 paper. Additionally, we will evaluate applicability of recently developed analytical techniques for developing smart mobile medical applications under dynamic context [14] application suites developed using this data sharing.

ACKNOWLEDGMENT

This research was funded in part by NSF grants CNS-1231590 and IIS-1116385. The work was done when Joseph Milazzo was in Arizona State University.

REFERENCES

[1] J. Milazzo, P. Bagade, A. Banerjee, and S. K. S. Gupta, "bHealthy: A physiological feedback-based mobile wellness application suite," in Proceedings of the conference on Wireless Health. ACM, 2013.

[2] S. K. S. Gupta, T. Mukherjee, and K. K. Venkatasubramanian, Body Area Networks: Safety, Security, and Sustainability. Cambridge University Press, 2013.

[3] A. C. Vivo, "Inter-app communication library," 2013. [Online]. Available: https://github.com/tapsandswipes/InterAppCommunication

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

[5] N. Agrawal, 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. San Jose, CA: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/Agrawal

[6] T. Gu, 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. IEEE, 2004, pp. 2656–2660.

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

[8] H. Kanoh, "Education for the net generation." [Online]. Available: http://www.childresearch.net/papers/digital/2008 01 01.html

[9] M. Persson, "Minecraft," 2014. [Online]. Available: https://minecraft.net/

[10] C.-K. Hsieh, 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, vol. 16, no. 4, pp. 6–7, 2013.

[11] A. Banerjee, S. K. S. Gupta, and K. K. Venkatasubramanian, "PEES: physiology-based end-to-end security for mhealth," in Wireless Health, 2013.

[12] F. Adelstein, G. Richard, S. K. S. Gupta, and L. Schwiebert, "Fundamentals of Mobile and Pervasive Computing", McGraw Hill, 2004.

[13] A. Banerjee, K. Venkatasubramanian, T. Mukherjee, S. K. S. Gupta, "Ensuring safety, security, sustainability of cyber-physical systems," Proc. IEEE, 100(1), 2012.

[14] A. Banerjee and S. K. S. Gupta, "Analysis of Smart Mobile Applications under Dynamic Co̅nte̅xt," IEEE Trans. Mobile Computing, Aug. 2014 (Early access publication).