# Developing Adaptive Quantified-Self Applications Using DynaSense

Pratik Lade, Yash Upadhyay, Karthik Dantu, Steven Y. Ko
Computer Science and Engineering
University at Buffalo, The State University of New York
Buffalo, NY, USA 14260
{pratikla,yashupad,kdantu,stevko}@buffalo.edu

***Abstract*—There are a number of user-centric applications that use data from sensors in a personal area network. The heavy dependence of such applications on sensors means that if a sensor is not available (e.g. a user forgets to carry a sensor device), some applications might not work properly or even fail. However, the data generated from a sensor that is unavailable can be derived from other devices or a combination of sensors. Since it is impractical and ineffective for application developers to track all such scenarios, user applications generally cannot take advantage of the sensor rich environment of a prospective user.**

**This paper introduces the design of DynaSense which is a middleware system that allows user applications to be agnostic to the data sources or sensors in use. DynaSense provides a unified approach for accessing data from various data sources, which can be sensors or compositions of other data sources. The middleware dynamically decides how to acquire data from available data sources, as well as how to deliver it to requesting user applications. We present the APIs that allow user applications to easily express their needs. We also present four case studies—a heart rate monitoring application, a user behavior anomaly detection application, a calorie tracking application, and a sleep monitoring application—to compare the development of these applications with and without DynaSense. These case studies show that DynaSense can effectively reduce the efforts of developers, in terms of the lines of code written.**

## I. Introduction

With the advent of smartphones, wearables, and Internet of Things (IoT) devices, sensing has become a ubiquitous part of our daily lives. Smartphones are capable of knowing our locations at all times; activity trackers are capable of counting the number of steps we walk everyday; smartwatches are capable of monitoring our heart rates; and the emerging smart clothing promises to measure the exact stress on our muscles at any given time. With all these devices, we are living in an era of being able to quantify various aspects of our daily activities at fine-grained time scales and round the clock. Numerous mobile applications are being built around these sensing modalities to provide us with nuanced information to characterize and improve various aspects of our lives. These applications are collectively called *quantified-self* applications; they can identify abnormal sleep patterns, help us reach exercise goals, remind us with to-do items based on our locations, etc. All of these applications are possible only because a user carries many devices equipped with various sensors.

While there are already many such applications, we make the following three observations on the current state-of-the-art quantified-self applications. The first observation is that each such application operates independently of the others processing the same data again on its own. The second observation is that while several applications are being enabled, the number of sensors/data sources are small, and so is the kinds of computation being performed on them. The final observation is that there is redundancy in the data collected; for example, both a smartwatch and a smartphone have accelerometer sensors which can be used to calculate the step count. But *most of the applications built are specific to sensors, rather than sensing modalities*.

Given these three observations, we have built *DynaSense*, a middleware that allows rapid development and deployment of quantified-self applications targeting sensing modalities while automatically figuring out best sensors to use. We have built this system on Android and demonstrate that building quantified-self applications is easy in DynaSense since we can leverage a large suite of libraries with common computations built in and improve their efficiency at run-time by the reuse of these computations. We also show that DynaSense is highly extensible by its very design, and can be used to build applications around novel sensing modalities such as smartwatches, smart clothing etc.

More specifically, the contributions of DynaSense are the following.

- **Design**: We propose a new design that frees quantified-self application developers from worrying about sensor management. Our design allows application developers to write their application logic with *data sources*, instead of specific sensors. A data source is essentially the data itself, e.g., a step count, a number of hours of sleep, etc., that can be acquired from either a single sensor or multiple sensors. Our design provides APIs to define a data source, as well as the acquisition and delivery of data from data sources to user applications.
- **Implementation**: We have implemented our design on Android as middleware that sits between data sources and user applications. Our middleware is essentially a publish/subscribe system that consists of a naming service, a publisher handler, and a subscriber handler.

- **Case Studies**: We have developed four applications with and without DynaSense to evaluate how effective our design is in reducing the efforts of developers. Our studies show that in terms of the lines of code written, DynaSense drastically reduces the lines of code written for user applications by refactoring sensor management from applications.

In the following sections, we will describe the design and implementation of DynaSense. Section II motivates the need for DynaSense. Section III discusses the design choices we made in DynaSense as well as our overall architecture. Section IV demonstrates the utility of DynaSense by redesigning existing monolithic applications in DynaSense and showing the benefits through code metrics and performance measurements. Section V discusses related work. Finally, we conclude in Section VI with our contributions and future work.

## II. Motivation

We first motivate our work in this section by surveying the class of applications we are interested in, and discussing the limitations of the state-of-the art.

### A. Quantified-Self Applications

Sensing in our daily lives has enabled a new suite of applications that were previously not possible. Tracking users provides more context about the user for marketing and advertising companies; a user's location alone can be used to suggest the nearest grocery store, coffee shop, shopping mall, gas station, and other location-specific information for more targeted advertising. Inference of personal context is useful for the user as well. It is not uncommon for athletes and surgery patients to manually track their vital statistics like weight, diet plans, sleep, exercise regimen, etc. However, with the advent of sensor-rich mobile devices like smartphones and smartwatches, self-tracking is becoming easier without the need for manual intervention. Even day-to-day activities can benefit by better-tracking activity levels, stress levels, sleep quality, calorie consumption, and so on. Wearable sensors provide the means to continuously monitor user data. A high rate of sampling of this data gives great insights into a person's life, and correspondingly a better quality of life in the long run. Given these possibilities, there has been an explosion of personal monitoring applications (or sometimes called *quantified-self* applications). We are particularly interested in this class of applications that are envisioned to enhance our daily lives. We can broadly classify them as follows:

- *Daily Activity*: An application can monitor calories consumed and spent, sleep quality and duration, stress levels, moods, etc. Most calorie expenditure applications typically rely on the accelerometer sensor on the wearable device, and uses that data to compute the various activities of interest.
- *Overall Health*: An application can track features like heart rates while exercising, asthma and diabetes levels. These applications use specialized sensors (e.g., for asthma), or customized algorithms (e.g., one can use the camera for measuring heart rate).
- *Context*: An application can maintain the user's location (e.g., work, home, playground) and the activity the user engages in (e.g., watching TV, sleeping, or exercising), and use this to trigger other measurement. Typically, such sensing uses a combination of the GPS, inertial sensors, and external information (such as WiFi hotspot, cellular location).

The takeaway from this classification is that most of these applications use a small subset of sensors. Typically these are, a way to infer location (GPS, WiFi, cellular radio), means to infer activities (accelerometers, gyroscope), and a modality to obtain a richer context (camera). Therefore, an efficient system would be able to provide data from these sensors to the appropriate algorithms and re-use them as and when required.

### B. Observations for Quantified-Self Applications

Given this class of quantified-self applications, we make the following observations. First, While the quantified-self movement dates back to the 1970's, miniaturization of sensing and computing has made them ubiquitous today. A smartphone for example is equipped with an accelerometer, light and pressure sensors, a compass, a camera, and a microphone, each of which can be used to measure and/or compute context of an individual. We also expect a smartphone to be carried around by the owner throughout her day. Similarly, wearable gadgets like smartwatches and smart clothes (that are closer to our bodies) add complex sensors like galvanic skin response sensors, heart rate monitors, $VO_2$ sensors and others to the above list. Finally external sensors that can be plugged into our personal area network (PAN) like blood pressure monitor or a weighing scale are also flooding the market at a rapid rate. Smartphones serve as a hub where the data from all PAN scale sensors can be collated and accessible for various applications.

This demonstrates the abundance of sensing modalities. However, there is another observation we would like to make, which is the redundancy of many sensing modalities. A microphone for example, is potentially present in a user's smartphone, laptop, smartwatch and intercom. Similarly, an accelerometer is present in a smartphone as well as a smartwatch. Ideally, quantified-self applications should be able to access the sensor data they require without hardwiring that to the particular sensor allowing them to have the flexibility to use any/all data available at any given moment. For example, a user going for a run might have forgotten to take his smartphone but is wearing his smartwatch. A calorie counting application should switch its input accelerometer data to use the data coming from the smartwatch, and not be tied to work with the accelerometer from the smart phone only. This example highlights the importance of inferring context (unavailability of the smartphone), and the ability of a runtime to appropriately switch the sensor data source as an important feature of future quantified-self applications.

Finally, like we mentioned earlier, the number of sensor values of interest is limited. This also limits the potential

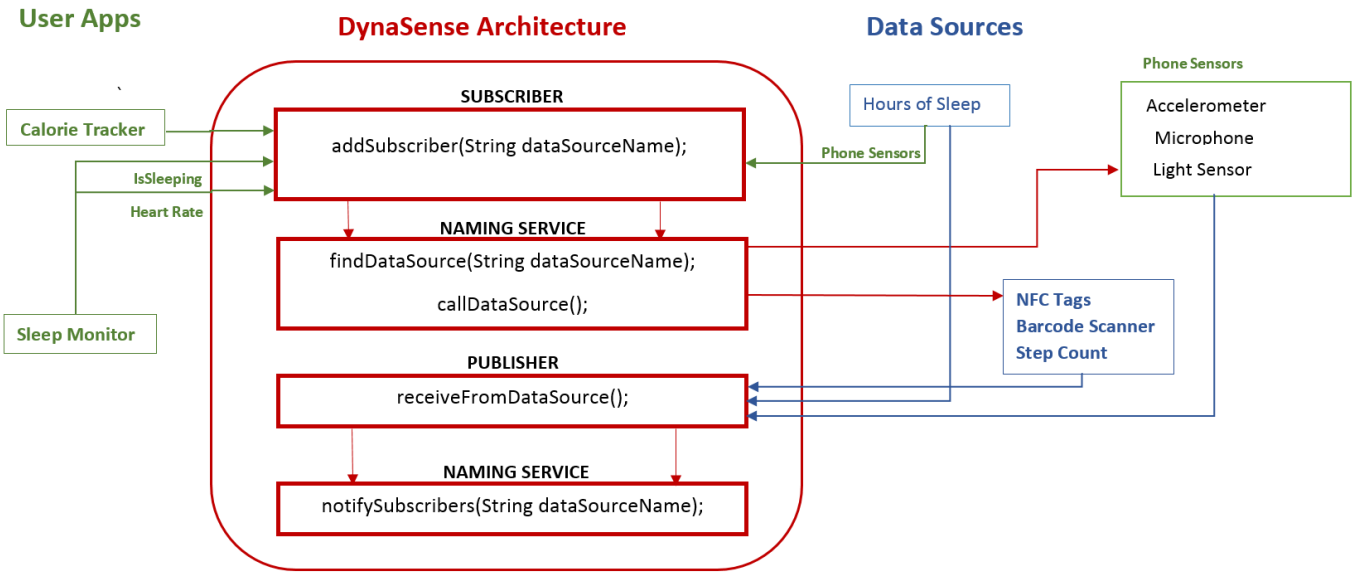


Fig. 1. DynaSense Architecture

information that we can infer from this data. However, each inferred piece of information (sometimes referred to as soft sensors) can be used in multiple ways by different applications. For example, a step counter can be used to calculate the number of calories burnt by the individual as well as to infer her activity level for the day. Both these applications however do not need to re-calculate the step count. An efficient system would allow re-use of such computation across applications to minimize the amount of redundant computing done.

### C. Summary of Our Motivation

Sensing and computing is becoming both ubiquitous as well as cheap. This has enabled a new class of personal monitoring or quantified-self applications that have the potential to revolutionize our daily lives. However, most such applications are being built monolithically without any overarching architecture. There are several features that would be desirable that do not exist currently because of this. Some of them are as follows:

- *Sensor Multiplexing*: The same sensor values are available from various devices. We should be able to use the best source for a given type of data at any given point without hardwiring an application to a particular sensing source
- *Reuse Sensor Computation*: Several computations we perform on sensed data is common across various applications. The programming model and runtime should allow for efficient re-use of computation across applications
- *Code Reuse*: Given that there is a lot of computation that is repeated across applications, a framework should provide the commonly used computations to simplify programming for the developers

Given these goals, we have built *DynaSense*, a framework that simplifies the programming of quantified-self applications in modern mobile systems.

## III. DynaSense: Design and Implementation

As discussed in Section II, we have two primary design objectives. We would like to allow applications to access sensor data without tying them down to a particular sensor. The second design objective is for the programming framework to allow efficient code re-use across applications for rapid application development. And finally, the third objective is for efficient re-use of computation at run time across applications.

Figure 1 shows the architecture of *DynaSense*. It consists of three main components—user applications, data sources, and the DynaSense middleware. *Data sources*, as the name suggests, are sources of data. These could be physical sensors in devices (such as accelerometers or gyros in smartphones and smart watches), or soft sensors that process data from sensors to produce data (such as step count or calories burnt from accelerometer data) for other applications. Both of them can act as producers of data. Applications are consumers of data. They take as input one or more types of data and produce useful information for the user (e.g., an application can take heart rate and step count to produce the person's stress level). The DynaSense middleware sits in between the applications and data sources, and connects the two at run time. The primary reason for this design is to be able to re-wire the connection between producers and consumers at runtime. The middleware tracks the user context, sensors available, and other relevant information at run time to make this happen.

In the rest of the section, we describe our DynaSense ecosystem in detail. It consists of APIs to interface applications to sensed information, and wire an application together. It also consists of a run time that runs as a service on the smartphone to track context and assist currently active applications. Finally, we will discuss how DynaSense meets all of our design goals.

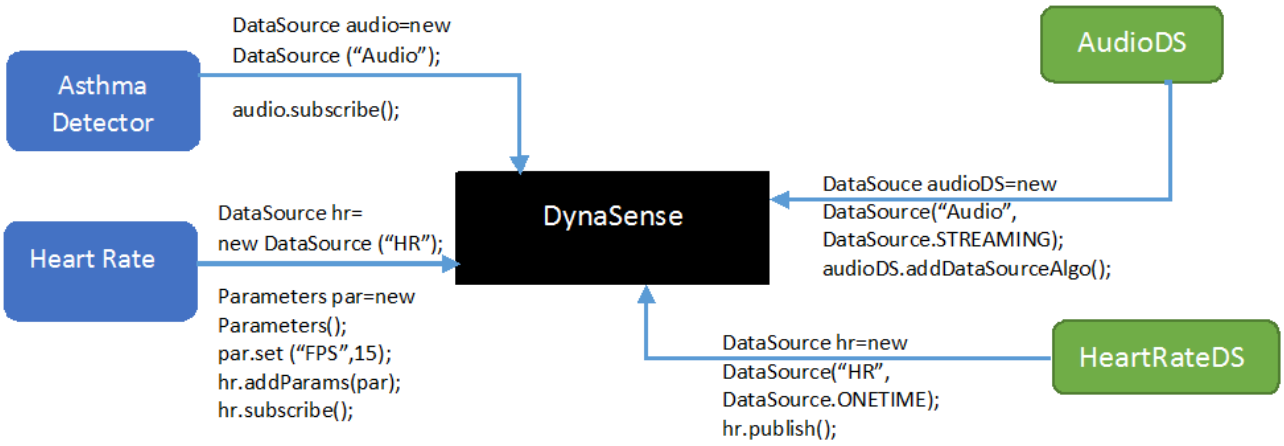


Fig. 2. Developer’s Interaction with DynaSense

### A. DynaSense Usage

Our envisioned scenario for DynaSense is that there would be regular application developers that write user applications, and data source developers that implement personal data analytics algorithms (e.g., a new step count algorithm). All of these would be available in an online application store (e.g., Google Play) so that end users can benefit from latest algorithm implementations by downloading new data source libraries similar to applications. The DynaSense middleware would be an application downloadable from an online application store as well, and is expected to run without any system privileges on the smartphone.

### B. DynaSense APIs for User Applications and Data Sources

As has been described in our design goals, we would like the applications to compute with sensed information, and not be directly tied to the sensors themselves. In order to achieve this, DynaSense uses a publish-subscribe system to connect the data sources to the applications. The publish-subscribe system is ideally suited for our purpose as the connection between publishers and subscribers is done at run time. The DynaSense framework provides APIs for user applications and data sources so that they can publish data, subscribe to data sources, or do both. The goal of the user application APIs is *ease of programming*; it simplifies how an application accesses sensor data by hiding low-level details about sensors such as where they are located and how they should be accessed. The goal for our data source APIs is *composability*, so that data source developers can provide new types of data easily. Table I lists our APIs. As seen in Figure 2, our APIs allow hierarchical structure for publishers and subscribers where a publisher can also subscribe to lower level data. The blue boxes represent user applications and the green boxes show data sources. The Heart Rate user application requests a heart rate from DynaSense which results in a call to HeartRateAlgo. HeartRateAlgo can use DynaSense to get camera frames with parameters like duration and number of frames. Thus HeartRateAlgo uses the Camera sensor as an input and produces a soft sensor value. When HeartRateAlgo is unavailable, DynaSense can call an alternate data source application that provides a heart rate.

The data source interface also comes with a naming convention to specify the type of data being published/subscribed. This is analogous to the mime types in a browser to recognize the particular file type. Further, the interface is also parameterized allowing the publisher to specify particular features of the data being published (such as resolution of the image or rate of the accelerometer data) and the subscriber to specify the application requirements (e.g., image resolution and data rate). As will be described later, the runtime resolves the publisher and subscriber parameters to ensure that the data publisher satisfies the application requirements efficiently (e.g., a sensor is sampled at a rate that meets all applications subscribing to that data will not exceeding them).

*1) Publishing:* A data source is an application that publishes a specific type of data. To illustrate, let us consider the example of the data source ‘Calories Consumed’. As part of the initialization, the application registers a data source of type ‘Calories Consumed’ with DynaSense. Our design assumes that both the developer of the publisher and subscriber application are aware of our data type name convention, and that it is consistent across applications.

Publishers can also specify parameters that describe characteristics of the publisher through this interface. Parameters for the publishing can be set by invoking the Parameters APIs. Once the parameters are specified, they are passed as an argument to the data source initialization.

```
DataSource ds = new DataSource("Audio");
Byte[] samples = getSamples(int samplingRate);
Bundle audioDetails = new Bundle();
audioDetails.putByteArray
```

TABLE I
DYNASENSE'S APIS FOR DEVELOPERS

| DataSource |
|---|
| dataSourceName:String<br>appName: String<br>appPackageName: String |
| addNewDataSource()<br>addParams(params: Parameter) subscribe(endTime: Calendar)<br>publish(details: Bundle) |

| Parameter |
|---|
| parameters: Map⟨String, String⟩ |
| get(param:String)<br>set(param:String, value:String) |

```
        ("AudioData", samples);
ds.publish(getApplicationContext(),
        audioDetails);
```

The above code declares to DynaSense a data source that provides audio data. Adding a new data source is functionally equivalent in DynaSense to declaring a new publisher. This information is captured by the DynaSense runtime, and kept track of. Data sources can be one-shot, periodic, or aperiodic. One shot, like the name suggests, publishes one reading. Periodic publishers publish data at a time interval specified by the subscribing application. Aperiodic data sources like step count are published at irregular intervals. This is mainly for data that might be triggered by an event, potentially external to the system, something that might not be of periodic nature or whose period cannot be determined by the runtime.

Publish requests are dispatched to the DynaSense runtime. Our current prototype is built in Android, and employs Android intents for this communication. However, this can be easily re-implemented through any other messaging service depending on the target platform.

The data source starts publishing data once it receives a request from DynaSense. To receive requests, a data source needs to maintain a list of applications requesting this data. In our current implementation, this is accomplished by extending BroadcastReceiver. The onReceive method of a data source parses the data source requested, and then starts further processing required to publish that data.

Applications can publish multiple data sources, but needs to declare each one of them to DynaSense. Hence it becomes important to parse the incoming request to identify the requested data source. The incoming request also includes parameters that were sent by the data user in a bundle.

```
DataSource ds = new DataSource
        ("CaloriesConsumed");
String[] fDetails = new String[]
        {foodName, calories, servings};
Bundle details = new Bundle();
details.putStringArray
        ("BundleValue", fDetails);
ds.publish(
        getApplicationContext(),
        details);
```

*2) Subscription:* A data user is an application that is in need of a specific type of data. In the case of a calorie tracker, an example of this is 'calories consumed.' To send a request to DynaSense, the applications subscribes to the data type "CaloriesConsumed."

```
DataSource calCon = new DataSource
        ("CaloriesConsumed");
calCon.subscribe(endTime);
```

Similarly, an application can subscribe to audio data and get data periodically. It specifies the duration for which the data is required and adds parameters like "SamplingRate" and "Channel" which are predefined by the publishing application.

```
DataSource audioDs=new DataSource(Audio);
Calendar endTime = Calendar.getInstance();
endTime.add(Calendar.SECOND,10);

Parameter audioParams=new Parameters();
audioParams.add(SamplingRate, 44100);
audioParams.add("Channel", "MONO");

audioDs.addParams(audioParams);
audioDs.subscribe(this.getApplicationContext(),
endTime);
```

### C. DynaSense Middleware

The DynaSense middleware is an application-level service that runs in the background. Its main job is to bridge user applications and data sources. In order to accomplish this, the middleware needs to perform four tasks as follows:

- Tracking what data sources are available
- Tracking the types of data the available data sources can produce
- Tracking applications and the requested data types respectively
- Delivering data to appropriate user applications when available

Below, we first describe the meta data that the middleware maintains to perform these tasks. We then describe how the middleware accomplishes the above objectives.

*1) Middleware Runtime Bookkeeping:* The DynaSense middleware maintains three types of information. First, it maintains the list of user applications and data sources. These are dynamically registered and the middleware monitors them for their availability. Second, the middleware maintains the list of available data types. For example if the data type 'StepCount' can be published by three applications, then the middleware maintains information about all three applications available to produce step counts along with the data sources that can produce it. Third, the runtime maintains a list of subscriptions. Note that each of the data types will potentially have multiple subscribers making it a one-many mapping. These are stored in a SQLite database in our implementation.

*2) Middleware Operations:* As mentioned earlier, there are four tasks that the middleware performs—maintaining the list of user applications, the list of data sources, the list of data types, and delivering data. For the first task (maintaining the list of user applications), the middleware responds to *subscribe* requests from user applications. This triggers a query in the DynaSense database for all matching data sources. If multiple data sources are found, the DynaSense runtime has the ability to pick a suitable one based on a prior policy (such as best resolution or most energy efficient). Currently, the first available data source is picked. If that data source returns bad values or times out, DynaSense marks it as unavailable and chooses another data source if available. If no other data sources are available, it sends a notification to the application that the data source is unavailable.

To maintain the list of data sources, the middleware responds to *registration* messages that a data source application sends to register itself. However, after the registration, since our system relies on data sources that connect to sensors in the personal area network, it is possible for some sensors to be unavailable. For example, a person forgetting to wear a smart watch when she steps out would result in an application exception for applications that connect to data sources on the smart watch. Such data sources could be temporarily out of service but may become a part of the system after some time. Also, it is possible to have data sources that always return bad values. This would be the case with data sources that are developed incorrectly or are malicious. We need to make a note of such data sources and mark whether they are temporarily or permanently out of order. The DynaSense service keeps track of such things by being the central point for all communication. All requests for context monitoring go to the service first so that it chooses the best data source to be contacted. Similarly, all data source values go to the service first so that if they are not received within a certain timeout period, the service can mark them unavailable and choose the next available data source. Additionally, this allows us to enforce policies for choice of data sources when more than one data sources can publish the same data.

To maintain the list of data types, the middleware first receives the information about what data types are provided by a data source in the registration process. However, if there are multiple data sources that provide the same data type, we need to keep the information and select one source when the data type is requested by a user application. In such cases, some policy of data source selection is required in the middleware. For example, consider a person who normally wears a personal fitness tracker that monitors step count. If at some point the person forgets to wear it, DynaSense should detect failure in fetching step count data from the fitness tracker, and automatically switch to the step counter data source that collects data from the smartphone instead of the fitness tracker. Our middleware currently uses a first-encounter policy, where the first data source registered for a data type is used to provide data.

Finally, to deliver data, a data source application uses the `publish()` method of the library to send it to the middleware. This triggers the `receiveFromDataSource()` method which queries the table `UserApp` to find all subscribers for that data source and forwards the data to each of them using the `notifySubscribers()` method. Thus, the DynaSense itself does not store any data.

To illustrate the working of the implementation of DynaSense, we revisit the example of a user forgetting to carry a smartphone on the user's daily run. In this case, we are assuming that the subscribing application, the middleware service and the publishing application that ultimately sends the step count to the subscriber all exist on the smartphone. In a typical scenario, the subscribing application monitors the step count of the user at frequent intervals throughout the day. In this situation, while the user is out for a run and both devices are not communicating with each other, the smartphone keeps reporting that there has been no increase in number of steps taken since morning. As soon as they get connected, the publishing application detects the smartwatch and fetches the latest step count. Since this functionality is built in the publishing application, the subscribing application on the smartphone did not need extra code to take into account the availability or lack thereof of the smartwatch at any point throughout its processing.

## IV. Evaluation

The primary goal for DynaSense is to simplify application development, and better enable better code and computation re-use. To demonstrate these features, we have implemented four applications— a heart rate monitor, a user behavior anomaly detector, a calorie tracker, and a sleep monitor. For each application, we have implemented two separate versions; one using the DynaSense framework and the other in Android without DynaSense. This section will compare these two versions to evaluate the usefulness of DynaSense. For each application, we compare the total lines of code written with and without DynaSense. We also demonstrate that running DynaSense does not add significant overheads to application run time making it feasible for most if not all quantified-self applications.

TABLE II
LOC (LINES OF CODE) COMPARISON OVER DIFFERENT APPLICATIONS

| | Heart Rate Monitoring | Sleep Detection | User Anomaly Detection | Calometer |
|---|---|---|---|---|
| Without DynaSense | 276 | 477 | 181 | 519 |
| With DynaSense | 160 | 196 | 153 | 119 |
| Data Source application | 131 | 183 | 81 | 403 |

### A. Case Studies

We have developed four applications with and without DynaSense, and compared the numbers of lines written. Table II summarizes the results from this effort. As the results demonstrate, using DynaSense has a tangible benefit in reducing the lines of code that need to be written. The use of DynaSense results in a high degree of code reuse by separating sensing from computing. The DynaSense library has 114 lines of code and the DynaSense service has 298 lines of code. In implementing our applications with DynaSense, we observed that a substantial amount could be reused. For the heart rate application, around 40 percent of the code was reusable after the camera data source was separated from the heart rate algorithm. For the anomaly detection application, the algorithm has been implemented in 315 lines, but the degree of code reuse is variable. In the current implementation a file is used as the data source, and the sensing component had 81 lines of code. However, a rough implementation of a sleep detection application has around 400 lines which means that while sensing actual real time context variables, there will be more than a 100 percent code reuse. Below, we describe each application and discuss what the differences are with and without using DynaSense.

*1) Heart Rate Monitoring:* The first application that we created using DynaSense uses a smartphone's camera to record images when a finger is placed on the camera lens [10], [7]. The idea here is that as blood flows into our capillaries, they increase in size obstructing more light around the camera lens. When blood flows out of the capillaries, their size reduces and they obstruct comparatively less light. We use these frames to calculate the average red component values or brightness values. These brightness values are then processed to remove outliers, smoothed over a fixed window and plotted against time to give a reading similar to an ECG. Heart rate is calculated using a simple formula: HR = (60 x frame rate x no of peaks)/ no of frames.

The first version of this application uses Android's API to access the camera and record about 300 frames at 15 fps. The code was divided into two parts: one for accessing the camera and the other to process brightness values. The latter part consisted of 160 lines of code but required a longer time for research and troubleshooting. The former mostly consists of code that can be used by other applications that use camera frames, such as face recognition. Thus, it is a good candidate as a data source application that produces camera frames.

With DynaSense, the implementation consists of a data source (CamDS) and a user application (HRCam). CamDS gives camera data. HRCam uses DynaSense to subscribe for data source "Camera." In CamDS, we first register it as a publisher:

```
DataSource camDS=new DataSource("Camera");
addDataSource(camDS,
              DataSource.STREAMING);
```

When HRCam needs to calculate the heart rate, it subscribes for "Camera" data source as follows:

```
DataSource cam=new DataSource("Camera");
Parameters camParams=new Parameters();
camParams.add("FPS",15);
camParams.add("TotalFrames","300");
cam.addParams(camParams);
cam.subscribe();
```

With this setup, a third party developer can create an application to record camera frames from the laptop and send them to the smartphone over Bluetooth/WiFi.

DynaSense can then choose to procure images from different sources without affecting HRCam which is unaware of the source of data. We were able to reduce the number of lines written from 276 lines to 160 lines in the user application. The data source application has 131 lines of code, that is reusable for other user applications. This simplifies development also by cleanly separating the data manipulation from sensor initialization and data acquisition.

*2) User Behavior Anomaly Detection Using DynaSense:* This application is a re-creation of previously published work on user behavior anomaly detection [4]. It is intended to demonstrate the benefits of DynaSense in an application that has already been developed. The application is intended to be useful for children or older people whose activities need to be monitored remotely. In the implementation, the following user activities are tracked: sleeping, meal preparation, washing dishes, entering, and leaving home and working. The current implementation uses a dataset of a resident living in a smart home for six months [1]. This application learns the normal behavior of a user for specific days of the week and uses it to detect anomalous behavior [4]. The idea is to learn the occurrences of different activities by creating multiple models for each activity.

To track a user's behavior, we need to track each activity of the user, the time it occurs, its duration and observe its relationships with other activities. To model individual activities, we use these features along with the total number of occurrences in a day and cluster them using DBSCAN which

is a density based clustering algorithm. After clustering, each activity occurrence is either classified into a cluster or noise. The clusters that are close to each other are merged to create a second layer of clusters that represent relationships between activities. Each new activity can now be classified as routine behavior or an anomaly.

To implement this application on Android, we created a module for the DBSCAN algorithm using existing libraries. Further, we created a class that represents activity instances and classifies them based on existing clusters. The dataset is saved as a file and so to get data, we need implement standard file operations. To develop this application for real world use, we need to gather data from several sources to find the user's current context. One of the activities that this application monitors is sleep. As described in the next section, a rough implementation of a sleep monitoring application has 400 lines of code. This tells us how large this user anomaly detection application would be if it implemented its own logic to monitor user's location, sleep and current activity. The contextual information that this application needs can be used by other similar applications too and so this is a classic example of code reuse.

The idea of using a dataset for such an application is to verify the design and correctness of the anomaly detection algorithm which is the most important part for the success of such an application. However, for real world use, this application would require a very high degree of sensing. As mentioned in the motivation for this thesis, the application developer would have to proceed in a depth first manner to track individual activities. For example, starting with sleep detection the developer would have to interface with all sources of data that a sleep detection algorithm needs. Similarly, to detect leaving and entering home, exercising or preparing meal the user would have to have create a lot of sensing modules. If DynaSense is used, the application can register for any number of activities that it wants to track and focus only on improving the anomaly detection algorithm.

To implement this application with DynaSense, we create an application called User Behavior which implements the DBSCAN algorithm. This application registers for data source "User Context." The data is provided from an application called "User Data" which reads data from a file and sends it to DynaSense. In a practical implementation of User Behavior, a host of applications monitoring the user's context would be created to publish data like "At work," "Driving," "Sleeping," etc. This would result in the application being very small as compared to its sensing components. Also, as new sensing components are developed, the granularity of context monitoring can be refined without changing the application logic of User Behavior.

*3) Calorie Tracking:* One of the major motivations for quantified-self applications is personal health. A major challenge in personal health has been to track the number of calories consumed by an individual. This application aggregates several techniques towards this goal.

Automating the measurement of daily calorie intake has not been perfected yet and smartphones do not have the capability to accurately measure the calories a person ingests. Hence external sources need to be relied on to monitor the intake of calories depending on food consumption. Services such as MyFitnessPal and Nutritionix store large databases of nutrition information online. This information is accessible through their APIs and allow us to look up the calories consumed (with detailed breakdown) given the product code of the item eaten. Nutrition information is also based on manual input if the food being eaten does not come with a product code. Finally, there are novel devices such as Vessyl[1] that allow for automatic calorie measurement of liquids, and can be interfaced with through Bluetooth.

The application also measures daily activity or expenditure of calories for which the user needs to manually log a workout session. The calorie lost for each activity such as walking, running, biking, etc. is then estimated.

As a standalone application, we provided calorie consumption tracking in three ways: using a barcode scanner to scan barcodes from food products, NFC tag identifiers to specify frequently consumed known quantity of food, or manual entry for everything else. We also added a feature to measure the number of steps walked taken daily.

To develop this application, we created a user interface which gives user the option of adding a new item using barcode scanning, by tapping an NFC tag or by manually entering data. This application can be separated into two parts where one part which includes the user interface registers for data sources "CaloriesConsumed" and "CaloriesSpent." This part of the application would fetch data from DynaSense and decide how to use it (for example, store it in a database). Data sources can be created that provide barcode scanning, interfacing with Vessyl or logging workouts that publish data source "CaloriesConsumed" or "CaloriesSpent."

With DynaSense, we separated the data collection from the data assimilation and analysis splitting the calorie intake and expenditure into several applications. We implemented an application for barcode scanning that searches an open health database for the product code scanned. It publishes the product information to DynaSense as the data source `Calories Consumed`. Similarly, we developed an application that can add new NFC tags and define what they signify(for example, a cup of coffee with 250 calories). Each of the features can be individually developed, tested and added to the ecosystem at any time with little to no changes to the application that does the analysis. This reduced the size of the application for calorie tracking to one fourth of its size when compared to the application without DynaSense. Such separation also allows for independent development of novel data sources without affecting the end application.

*4) Sleep Monitoring:* Toss n' Turn[8] describes an approach to detect sleep and monitor sleep quality. A similar approach to monitor sleep is also described in Unobtrusive sleep monitoring[3]. We used these references to implement

[1]https://www.myvessyl.com/

a sleep monitor that uses various sensory inputs to detect whether a person is sleeping at the moment.

To detect whether a person is sleeping, the most important factors can be intuitively guessed, such as: when was the phone display on, how much is the surrounding noise, what is the ambient light level, has the phone been moving, and so on. We used factors that were considered in Toss n' Turn[8] and built the sleep monitoring application.

We built a simple UI that shows whether a person is sleeping or not at any given time. Data from ambient light sensor, accelerometer, step counter, battery charging state and device screen state is collected through Android's Sensor API and given as input to a decision tree to conclude whether a person is sleeping. This application can be separated into two modules: the application that users interact with which consists of a UI that displays sleep duration and quality, displays historic data, etc. and an application that monitors phone sensors periodically and inputs them to a decision tree algorithm for classification.

In the implementation in DynaSense, the sleep data source application takes the period at which sleep should be monitored as a parameter. It evaluates if the user is sleeping when data from all sensors is received. This information is published as a data source `isSleeping`. The sleep detection application uses this data source, and stores historic sleep data to show statistics to the user. By separating the two applications, there is a 50% reduction in the lines of code for developing the application without DynaSense. Also, other applications can make use of the sleep data source for other purposes. Further, other sources like smart watches can provide sleep data to DynaSense dynamically without changing the user application that needs this data.

Since the algorithm heavily relies on readings coming directly from sensors, there is not much scope for overlapping data sources. The device battery state and screen state are unique sensors that cannot be replaced by anything similar.

### B. Performance

Although the focus of DynaSense is the ease of programming and maintenance for application developers, we present performance results that demonstrate the fact that DynaSense does not impose much overhead. We have measured the performance of DynaSense in three aspects—the subscription latency of a source or an application, the lookup latency within the DynaSense middleware, and the end-to-end latency for delivering data. We have used an LG Nexus 5 device running Android 5.1. For each data point, we report the average and the standard deviation over 10 runs.

*1) Subscription:* The first bar in Figure 3 shows the subscription latency. This is a one time cost that incurs when a user application first registers with DynaSense for a data source, which takes around 15 *ms* on average. Although this operation is asynchronous, a user application does not receive any data until its subscription for a data source is completely finished. Thus, this time lag is negligible if the user application is requesting periodic data since this initial time gets amortized over time. However, for applications that request one-time data sources frequently, this latency will contribute to the time required for their overall data access latency.

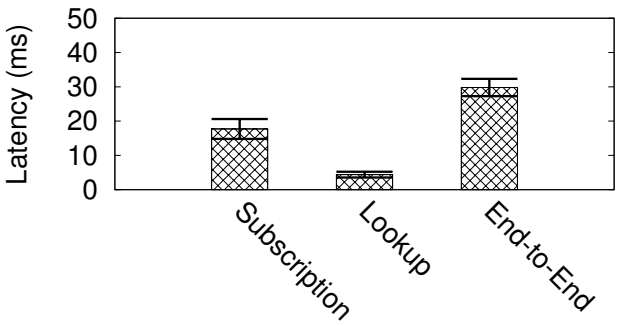


Fig. 3. DynaSense Microbenchmark Results

TABLE III
COMMUNICATION AMONG PROCESSES

| No. of Intents | Initial Subscription | Publishing | Total per Transaction |
|---|---|---|---|
| One Shot | $1+1$ | $1+1$ | 4 |
| Periodic | $1+1$ | $n \times (1+1)$ | $2 \times n + 2$ |
| Aperiodic | $1+1$ | $n \times (1+1)$ | $2 \times n + 2$ |

*2) Lookup:* The second bar in Figure 3 shows the lookup latency within the DynaSense middleware. When a data source sends a data item to DynaSense, it retrieves the user applications that have subscribed for that data source. For example, when the camera data source sends frames to the middleware service, it retrieves Heart Rate application as one of the subscribers. We measure the latency for this lookup, which is around 4 *ms* on average. We note that this is largely dependent on the specific implementation for lookup, hence we do not measure it in a more thorough manner; namely, we use SQLite that Android provides, and the lookup performance is exactly the lookup performance of SQLite on Android.

*3) End to End Performance:* The last bar in Figure 3 shows the end-to-end latency measured with a simple step count application that we have written. The reason why we have written a separate application is that the step count application does not have any complicated logic, hence is simple to measure the end-to-end latency. This step count application fetches the number of steps walked from the start-up time of the smartphone, whenever its user presses a button. As shown in Figure 3, the latency from the moment the user presses the button to the moment it receives the value averages to 29 *ms* over 10 runs, which is nominal for personal analytics applications DynaSense supports. This latency includes the latency for subscription and actual data delivery.

*4) Communication Between Processes:* Table III shows the cost of communication in DynaSense. Our implementation uses *intents* on Android, which is an IPC mechanism that allows communication between different applications. We use the number of intents as a unit to measure the cost of com-

munication. If a user application has a one-shot task that runs only once for a sensor value, the number of intents exchanged between a user application and DynaSense is four in our implementation. On the other hand, if a user application has a periodic or aperiodic task that runs continuously, the initial subscription process takes a total of two intents, followed by two intents for every time data is published. In addition to this, each new publisher needs to send one initial registration intent to the DynaSense service.

## V. Related Work

Context-awareness in mobile systems has received a fair amount of attention in research. In particular, there is work that focuses on efficiently allocating sensor resources in a mobile system based on context very much in the flavor of our work. We will describe four such systems in detail. We will also identify the primary differences between our work and these for clarity.

ODK Sensors [2] provides control of external sensors through *sensor drivers* that are written as mobile applications. The concept of sensor drivers is similar to our concept of data source applications in that both are written as mobile applications and control sensors. However, a sensor driver in ODK Sensors is similar to a traditional device driver where the primary job is controlling device hardware; our data source application is designed to implement new algorithms that synthesize multiple data sources, whether it be raw sensors or other data source applications. Our data sources are sensor-agnostic, and depend on the sensed data type allowing us to use the "best" available sensor at any given time that provides that sensed data type.

ACE [9] is a rule-based context inference system that uses multiple context variables to infer the values of other context variables. For example, if the context variable *atHome* has been computed recently and is true, an application requesting other context variables such as *isDriving*, *atWork*, etc. will automatically be returned false. Such rule-based inference is complementary to our approach where the primary focus is not to infer a context but to provide a general architecture that maintains and multiplexes sensors and algorithms that can produce high-level data such as contexts.

SeeMon [5] is an energy-efficient context monitoring system that continuously monitors the context of a user. In doing so, it leverages multiple sensors available within the user's personal area network. Similar to DynaSense, SeeMon provides an API that an application can use to query which context its user is currently in. The API also handles algorithms that produce higher-level data from raw sensor data, called context translation maps. This is roughly analogous to our data source applications that implement personal data analytics algorithms and produce higher-level data such as heart rate. However, unlike DynaSense, SeeMon has two limitations. First, it does not allow multiple data-producing algorithms to co-exist if they produce the same type of data. Second, it does not allow hierarchical composition of higher-level data sources from lower-level data sources.

Orchestrator [6] extends SeeMon to relax the first limitation. It is described as a resource orchestration framework, selects from multiple *pre-defined* plans for resource use, and selectively applies them based on resource availability and demands at run-time. The primary difference between Orchestrator and Dynasense is that the plans in orchestrator are pre-defined. This limits the expressivity of programs and requires the programmer/planner to know all available resources at design time. In contrast, our programming model does not bind a data source to a sensor, and makes this connection at run time. This allows for more flexibility including the use of newer sensors, and learning of context that over time for more efficient operation. Our contribution is a novel programming model that gets rid of the limitations in prior work by making applications address data as opposed to sensors.

## VI. Conclusions and Future Work

The goal of this paper is to design a programming model and runtime allows programmers to not worry about the available sensors but focus on the sensor data for the intended application. The DynaSense programming model achieves this by providing an abstraction called data sources and using a runtime to bind a data source to a sensor. By using DynaSense, user application developers need not concern themselves with writing tedious code to access data from individual sensors. At the same time, applications automatically support new sensors without changing any code in their application by just adding a new data source. We show that the overhead of the middleware is minimal, and our programming model is ideally suited for quantified-self applications by designing four applications. Our programming model also allows for greater code re-use.

In the future, we plan to explore various policies to decide on the best data source to use when multiple sensors are publishing the same data. We envision designing rich policies that trade off energy, quality of service, and computation depending on the exact application. For example, it would be interesting to explore battery optimization policies that can be used here to drive the choice of data source for an accelerometer data when accelerometers are present in a smart watch as well as the smartphone. In addition, we plan to expand integration of external sources such as smart glasses, pedometers and others to leverage their processing power as well as sensors.

## References


[1] Wsu casas smart home project, January 2011.

[2] Rohit Chaudhri, Waylon Brunette, Mayank Goel, Rita Sodt, Jaylen VanOrden, Michael Falcone, and Gaetano Borriello. Open data kit sensors: Mobile data collection with wired and wireless sensors. In *Proceedings of the 2Nd ACM Symposium on Computing for Development*, ACM DEV '12, pages 9:1–9:10, New York, NY, USA, 2012. ACM.

[3] Zhenyu Chen, Mu Lin, Fanglin Chen, N.D. Lane, G. Cardone, Rui Wang, Tianxing Li, Yiqiang Chen, T. Choudhury, and A.T. Campbell. Unobtrusive sleep monitoring using smartphones. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2013 7th International Conference on*, pages 145–152, May 2013.

[4] Enamul Hoque, Robert F. Dickerson, Sarah M. Preum, Mark Hanson, Adam Barth, and John A. Stankovic. Holmes: A comprehensive anomaly detection system for daily in-home activities. In *Distributed Computing in Sensor Systems (DCOSS), 2015 International Conference on*, pages 40–51, June 2015.

[5] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Youngki Lee, Souneil Park, and Junehwa Song. A scalable and energy-efficient context monitoring framework for mobile personal sensor networks. *Mobile Computing, IEEE Transactions on*, 9(5):686–702, May 2010.

[6] Seungwoo Kang, Youngki Lee, Chulhong Min, Younghyun Ju, Taiwoo Park, Jinwon Lee, Yunseok Rhee, and Junehwa Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 135–144, March 2010.

[7] Sungjun Kwon, Hyunseok Kim, and Kwang Suk Park. Validation of heart rate extraction using video imaging on a built-in camera system of a smartphone. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, pages 2174–2177, Aug 2012.

[8] Jun-Ki Min, Afsaneh Doryab, Jason Wiese, Shahriyar Amini, John Zimmerman, and Jason I. Hong. Toss 'n' turn: Smartphone as sleep and sleep quality detector. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 477–486, New York, NY, USA, 2014. ACM.

[9] Suman Nath. Ace: Exploiting correlation for energy-efficient and continuous context sensing. *Mobile Computing, IEEE Transactions on*, 12(8):1472–1486, Aug 2013.

[10] Panagiotis Pelegris, K. Banitsas, T. Orbach, and Kostas Marias. A novel method to detect heart beat rate using a mobile phone. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 5488–5491, Aug 2010.