ManySense: An Extensible and Accessible Middleware for Consumer-Oriented Heterogeneous Body Sensor Networks

Joonas Westlin and Teemu H. Laine

Department of Information and Computer Engineering, Ajou University, Suwon 443-749, Republic of Korea

Correspondence should be addressed to Teemu H. Laine; teemu@ubilife.net

Received 8 January 2014; Revised 11 June 2014; Accepted 13 June 2014; Published 10 July 2014

Academic Editor: Qi Han

Copyright © 2014 J. Westlin and T. H. Laine. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Recent advances in ubiquitous technology have enabled a new wave of consumer-oriented wearable sensor devices that monitor different aspects of a human body such as heart rate, temperature, perspiration and motion. Several global consumer technology companies have embraced the idea of body-awareness that could finally fulfill Mark Weiser’s vision of truly ubiquitous computing [1] from the human body perspective. There are many off-the-shelf products that can be used by anyone for personal body monitoring. Current trend of smartwatches and activity trackers is an example of this, but it is merely a preview of what future wearable technologies could be. Most smartwatches and other wearable technologies possess useful features such as mini applications, sensors, and wireless communications, yet they have not become killer devices of ubiquitous computing.

A combination of wearable sensors can form a body sensor network (BSN) for gathering versatile information on bodily functions. Researchers have applied BSNs especially in healthcare [2–8], but also other domains have been researched such as sport and fitness [9–12], transportation [13], social networking [14–16], and gaming [17–19]. One challenge in many previous BSN systems is that they support data collection through a fixed sink node using a specific protocol such as ZigBee. Such setup is based on a homogeneous set of compatible devices. Combining data from heterogeneous devices such as smartphone, smartwatch, and heart rate monitor can enable a higher degree of body-awareness, but it bears a cost of data aggregation complexity. Another challenge is that body sensors used in research projects are often prototypes or highly specialized medical sensors that are not easily available to the end users. On the contrary, the end user might possess heterogeneous off-the-shelf devices from different vendors that are not compatible with each other. To illustrate what these challenges mean in practice, let us consider a sport enthusiast who owns a Zephyr heart rate monitor, a Sony smartwatch and a...
smartphone. Currently, he has to use different vendor-specific applications on his smartphone to access each device's data at runtime. The situation gets even more complicated if the user also wants to follow his physical location using his smartphone's GPS. Assuming that the user has experience with software development, he could build a smartphone application that could gather data from each device and process it as appropriate. However, this is not a flexible solution because upon device replacement or addition the user would need to modify the software to accommodate new data sources. Furthermore, other users would have to repeat the same steps in order to develop software for their personal devices.

The aforementioned scenario illustrates the need for middleware architecture that is able to aggregate and process wearable sensor data from heterogeneous sources and provide access to the data for other applications, such as games, medical monitors, and performance analyzers. The data provided by the middleware should be available in multiple modalities from raw data to filtered data to context data carrying semantic value such as the user's activity. This way the third party application developers can choose an appropriate modality for their applications. The middleware should also allow flexible addition of new devices, sensors, and context inferencing algorithms with minimal changes to other components. Following these requirements, to solve the challenges of previous BSN systems, we propose ManySense BSN middleware for Android devices that not only supports multiple heterogeneous off-the-shelf wearable sensors but can also be extended by adding adapters for new sensors and context inferencing algorithms. ManySense provides a unified access to (a) raw sensor data retrieved from both internal and external sensors and (b) high level context data which is based on the inferential analysis of raw data. It also has a filtering capability for improving raw data quality by, for example, removing noise. ManySense can be used by any Android-based application that requires high quality sensor and context data from heterogeneous sources.

We start by describing related work on body sensor networks and middleware. After justifying the need for ManySense, we present its design and implementation with detailed description of the adapter interfaces that make ManySense extensible. We then evaluate and analyze ManySense from three perspectives. First, its performance (load handling, CPU, RAM, and power) is measured. Second, the adapter interfaces’ flexibility is analyzed by discussing the process of creating new adapters. Third, ManySense’s accessibility is analyzed by coupling it with Calory Battle AR—an augmented reality exergame—and utilizing sensor data to implement a new game activity. Implications of the study are then discussed and conclusions are made.

2. Background

2.1. Body Sensor Networks. In this paper, we propose the ManySense body sensor network (BSN) middleware to be used in context-aware Android applications that require heterogeneous data sources such as wearable sensors. A BSN (sometimes referred to as body area network (BAN)) typically consists of small, portable, wireless, and energy-efficient sensor nodes that are deployed on or inside a human body. The types of sensors depend on the target application. For example, in healthcare applications a BSN might consist of sophisticated medical sensors such as ECG, SpO2, blood pressure, and blood sugar, whereas an athlete might use a BSN with accelerometers, pressure sensors, and a heart rate meter. A common goal for most BSN-based systems is that they enable unobtrusive collection, analysis, distribution, and management of body signals regardless of time and place.

BSNs based on ZigBee or Bluetooth have been widely used for bodily context detection in health and wellness applications. A wireless sensor network developed in the Codeblue [3] project provides routing, naming, discovery, and security for wireless medical sensors, PDAs, PCs, and other devices that may be used for monitoring and treating patients in network environments of various densities. In-home monitoring [4] is a ubiquitous healthcare system for real-time monitoring of patients’ locations with GPS and their vital signs with wearable ZigBee sensors. ECG (electrocardiogram) monitoring systems [5, 6] aim to monitor electrical activity of a human's heart in real time with an ECG sensor, an ECG console, and a ZigBee module. ECG data collected from a sensor network is transmitted to a server through a gateway. In many BSN systems, a sink node is coupled with a Bluetooth module for communicating with a Bluetooth-enabled gateway device. LifeGuard [20] is a vital sign monitoring system for astronauts. It uses a wearable wired sensor kit that communicates with a Tablet PC via Bluetooth. In another example [21], a mobile phone acts as a gateway which connects between sensor nodes and a CDMA network or other devices having Bluetooth support. Viswanathan et al. [22] proposed a distributed resource provision framework in which nearby computing devices (laptops, tablets, and PDAs) use Bluetooth or ZigBee to collect and preprocess data before sending it further.

Based on our analysis of existing BSN systems, typical BSN middleware architecture is based on a layered model (Figure 1) comprising four or more layers. The BSN layer takes care of context sensing using a variety of sensors which measure properties of a human body such as movement, heart rate, or oxygen saturation. Data collected by the BSN layer is transported to the Gateway layer over a short-range communication protocol such as ZigBee or Bluetooth. The Gateway layer is represented by devices of limited processing power such as smartphones, laptops, or set-top boxes. These devices operate as gateways aggregating and preprocessing (e.g., noise filtering) data before sending them over the internet to the Backend layer. The Gateway layer may cache the data locally for quick access, visualization, data analysis, and context inference. Some gateways are also capable of sending control messages back to the BSN nodes. The Backend layer acts as a central repository for data collected from multiple deployed BSNs. It may provide advanced data analysis tools and interfaces for the Presentation layer. The Presentation layer offers multiple means of accessing, visualizing, and manipulating data by stakeholders including but not limited to users, healthcare professionals, and sport coaches. Data
compression and encryption techniques can be applied to interlayer communication to reduce required bandwidth and protect privacy.

2.2. Middleware for BSNs. Middleware is a crucial element in any distributed system for providing abstraction of low-level programming routines and for unifying access to heterogeneous services and resources. While many existing BSN systems are based on homogeneous and specialized sensor nodes, there have been attempts to create middleware supporting heterogeneous off-the-shelf sensor devices. KNOWME [7] aggregates data from off-the-shelf ECG, accelerometer, GPS, and SpO₂ sensors using the Nokia N95 smartphone. The system estimates energy expenditure by detecting user activity through feature extraction from sensor readings. Activity detection can be accomplished either inside the middleware or on a remote server. KNOWME supports only a limited set of sensors for healthcare purposes and to our knowledge, it has no flexible structure to add new sensor device types. Furthermore, there is no way for third party applications to subscribe for receiving sensor data from the middleware. Finally, N95 is an obsolete platform today. Park and Pak [8] proposed an integrated gateway architecture to collect data from various personal health devices (PHD, ISO/IEEE 11073) using a range of communication methods such as ZigBee, Bluetooth, and USB. The system only supports standard PHD devices and has no way of mediating data to other applications.

Aforementioned middleware are nonadaptive because they are limited to a specific set of sensors. To overcome this limitation, researchers have created adaptive middlewares which facilitate extensibility by supporting addition of new sensor devices. A lightweight adaptive BSN middleware is proposed in [23] for medical applications. This middleware supports dynamic addition of new sensor nodes based on TinyOS, security features, and on-the-fly sensor reconfiguration by control messages. The architecture is divided into lower middleware comprising wireless sensor nodes (nesC) and upper middleware which runs on a PDA (Java 2 Micro Edition, J2ME). The authors created a prototype implementation of the middleware which neither supports heterogeneous sensor devices nor allows other applications to subscribe to sensor events. Furthermore, J2ME platform on PDAs is obsolete today. MiddleWhere [24] is an adapter-based middleware to solve the problem of heterogeneity and extensibility. It uses multiple sources of location information in order to determine the user’s indoor location accurately. Adapter interface allows different sources of location data to be coupled with the system. While this approach makes the system extensible, it is limited to a single purpose. As the last example, SIXTH [25] is an Android-based middleware that takes a flexible approach to middleware design by utilizing the standardized OSGi (open services gateway initiative) component framework as the platform. OSGi allows new modules to be added at runtime, such as sensor adapters which extract data from different sensors. To add a new sensor type, a new adapter must be created and deployed to the OSGi service (Apache Felix) running on Android. If a third party component wishes to receive sensor data updates, it must implement an interface and register with the OSGi service. In addition to sensor data retrieval, the SIXTH middleware also supports retasking through sending control messages to sensors. Utilizing OSGi makes the system highly extensible, but it bears a cost of performance overhead and possible instability. Furthermore, only OSGi modules implementing a required interface can receive sensor data, thus leaving out Android systems which do not have the OSGi service running.

In addition to collecting data from heterogeneous sources, an important function of a BSN middleware is the inferencing of higher level context data by fusing, transforming, and processing raw sensor data. A typical example of higher level context data in the case of BSNs is the user’s activity which, once inferred, can be queried by third party application developers. Recognizing the user’s activity typically involves raw data from sensors such as accelerometer and GPS that are submitted to an inference algorithm which depends on the target application. For example, the detection of whether the user is walking, running, or sitting based on accelerometer data typically involves four stages: data collection, preprocessing (e.g., noise filtering), feature extraction, and classification [26]. If higher level of context data using a greater number of data sources is desired, a more sophisticated modeling and inferencing framework is needed, such as object-oriented models, logic-based models, or ontologies [27].

There exists several BSN middlewares that offer some level of context inferencing. ACE is an energy-efficient middleware for Windows Phone that focuses on context inferencing [28]. High level context data is inferred from raw sensor data by rules. Other applications may use ACE to request context data such as “IsDriving” or “AtHome.” As ACE focuses on context inferencing, it does not support heterogeneous wearable sources. Lara and Labrador proposed an Android-based system for real-time human activity recognition using phone GPS, phone accelerometer, and Zephyr’s BioHarness [29]. The user’s activity is inferred by the four-stage process as described above. The system uses the authors’ MECLA library which contains several
classification algorithms. Gu et al. [30] proposed a service-oriented context-aware middleware (SOCAM) which uses ontology modeling to provide context inferencing services for applications in home and vehicle environments. SOCAM’s inferencing logic is implemented on a server which acquires raw data from physical sensor devices and external virtual sensors such as web services. Other applications (e.g., mobile and desktop) can request higher level context data from the middleware. As the last example, myHealthAssistant is a smartphone application that uses inertial body sensors to monitor the user’s daily activities and gym exercises [12]. Activity detection is based on an event-based middleware running on the smartphone [31]. It supports heterogeneous sensor devices over HTTP and Bluetooth connections. Raw sensor data is transformed to events which are then used by the central context inferencing module that detects the user’s activity. The authors claim that the middleware has a modular architecture, but it is unclear whether alternative context inferencing modules can be installed.

Previous BSN middlewares have some shortcomings including lack of support for heterogeneous sensors, support for none or single context inferencing technique, lack of preprocessing tools such as data filtering, lack of a unified interfaces for third party developers to access raw sensor data and inferred context data, dependencies to other software, and obsolete platforms. In the following we describe the design and implementation of ManySense BSN middleware for Android which alleviates the challenges found in existing systems by not only allowing easy addition of heterogeneous sensor devices and context inferencing algorithms through adapter interfaces but also providing a simple access to raw sensor data and inferred context data for any third party Android application running on the same device.

3. System Design and Features

To keep the ManySense design flexible for future needs, our definition of ”sensor,” which follows the work of Gu et al. [30], does not only cover hardware-based physical sensors but also virtual sensors which include fusion sensors (e.g., gravity, linear acceleration, and orientation) and external data sources (e.g., web services and other online data repositories). This broadened view of sensors allows ManySense to be used in future applications that go beyond body context towards environmental and social awareness.

There were several principles that were followed throughout the design process. First, the middleware should be extensible so that adding new physical and virtual sensors as well as context inferencing algorithms would be easy. This is to make the middleware available for multiple purposes. Second, the middleware should encapsulate the implementation of data acquisition and context inferencing behind a common interface. Third, other applications should be able to access the middleware to retrieve data in multiple modalities including raw, filtered, and inferred data. Fourth, the middleware should be able to handle multiple requests from different clients at the same time efficiently. Fifth, the middleware should be able to change its configuration based on the presence of sensors, but application developers should also be able to control the middleware. Sixth, the middleware should provide tools (e.g., filters) for preprocessing raw data. Seventh, the middleware should provide data caching and remote storage services for further analysis and sharing. Finally, the middleware should be minimally dependent on other software. We utilized well-known software design patterns (observer, singleton, adapter, template method, and façade) to conform to these principles as well as to keep the system design encapsulated and loosely coupled [32].

Figure 2 illustrates the architecture of ManySense. In the center of the figure is the middleware which runs independently in an Android background service to which the subscriber application must bind itself. The ManySense service is only alive if there are subscribers bound to it, thus reducing battery consumption when idle. Because ManySense runs as a background service, it does not have a user interface except for a preferences screen that can be used to adjust parameters such as default intervals at which sensors send updates, time-out times, or user names and passwords required to access online sensor sources. At the bottom is the subscriber application which communicates with the middleware through a communication interface. To make the middleware easily accessible to application developers, the communication interface handles the binding details, so the developers do not need to be concerned with it. On the ManySense side of the communication interface lies the subscription handler which processes all applications’ requests and directs them to the correct component (context data aggregator or raw data aggregator).

Raw sensor data is acquired through the raw data aggregator (RDA) which collects data from heterogeneous sensors and distributes them to the subscribers. It communicates with different sensors through Sensor Adapters. When a Sensor Adapter receives new data it publishes sensor events to the RDA which in turn forwards the data to the subscribers which are subscribed to that particular sensor’s data. The RDA can also send data to a remote server for long-term storage and analysis.

The RDA contains data filtering algorithms that can be used to reduce noise, drift, or offset in the data, for example. These algorithms (e.g., low-pass and high-pass) can be optionally applied to raw sensor data before it is sent to the application. There is support for multiple filter implementations for the application to choose from, and filters may have control parameters such as the smoothing factor in some low-pass filters. Integrating optional filtering functionality into the middleware removes yet another task from the application developer.

Sensor Adapters offer a common interface for the RDA to request data from both physical and virtual sensors. This way the individual adapter implementations are encapsulated and the RDA does not need to be concerned about them. The common interface makes it easy to add new Sensor Adapters, thus improving extensibility. There exists one adapter for each supported sensor and each adapter holds an algorithm for acquiring data from its adapter sensor. Algorithm parameters such as default sensing interval or timeout can be adjusted through per-adapter preferences that are accessible through
Virtual sensors

| Physical sensors | Virtual sensors |
|------------------|-----------------|
| Onboard sensors  | Fusion sensors  |
| Smart watch      | Web services    |
| Heart rate monitor| Weather data   |

---

**Establish connections (Bluetooth, HTTP etc.), request data, Control messages**

Sensor data

Save and use global preferences

Preferences

**Default update delay**
Set the default delay between updates that should be used if no delay is set.

**Default timeout**
Set the default timeout after which requests fail.

---

The role of the context data aggregator (CDA) is to provide higher level context data to subscriber applications. It delegates the responsibility of context inferencing to Context Inferencing Adapters (CIA) which can be selected based on the application’s needs. Each CIA is responsible for subscribing to necessary raw data from the RDA and then processing it appropriately. Encapsulating the CIAs behind a common interface follows the same design that we applied in raw data acquisition; thus, it has similar extensibility. Examples of CIAs could be a physical activity detector using accelerometer data and an ontology-based adapter for detecting higher level activity such as “working,” “sleeping,” and “eating” using multiple sensors. As the complexity of context inferencing may vary significantly between various approaches, the CIAs can also send inferencing requests to a server. This would be essential in the case of large ontology-based context models, for example.

To facilitate the development of Sensor Adapters and Context Inferencing Adapters by third party developers, we have created an Adapter SDK (software development kit) that includes simulators, code examples, and documentation to assist in building custom adapters. The simulators take the roles of RDA and CDA by loading adapter components
and sending test requests to verify the adapter’s operation. Using the Adapter SDK, third party application developers can create and test new adapters for acquiring raw data as well as inferring context data. Currently ManySense does not support runtime deployment of custom adapters, but they must be inserted into the system by us. Nevertheless, the SDK facilitates adapter development and our future plan is to allow third party adapters to be attached to ManySense at runtime.

Subscriber applications communicate with ManySense either by method calls or by sending requests using ManySense query language (MSQL). Examples of these queries include selecting target sensor data types or context inferencing algorithms, setting sensing interval, applying data filters, and setting data collection schedule.

Collected and inferred data can be cached in a local Android database by the RDA and the CDA, and they can also be stored and/or analyzed on an external server. The cached and stored data can be used for data visualization and deeper analysis, such as ontology-based context modeling.

In order to demonstrate the extensibility and accessibility of ManySense, we can consider different user scenarios as follows. John is working as a team leader in an IT company. Apart from computers, his passion is running marathons and to facilitate training he owns a range of gadgets including an Android smartphone, Zephyr BioHarness body monitor, Pebble smartwatch, and Nike FuelBand. Until now John has used separate applications for monitoring each device. As a decent programmer, however, he can now write a simple application that uses ManySense to get data from all devices and plot data nicely on a single view. John’s wife Jane is a researcher in the field of educational technology. Her current research is about building a context-sensitive language learning tool that would provide situation-dependent language drills to users. Jane has little programming skill, but she is mainly focusing on user interface development and has no idea about sensors and such low level details. Before John told Jane about ManySense, she was planning on hiring an experienced programmer (or using her husband) to create a custom context-awareness module for her learning tool. Instead now she can use one of ManySense’s context inferencing algorithms to acquire high-level context and use that in her application to deliver context-sensitive learning content. Jane’s cousin Mike is a junior lecturer at a local university’s computer science department. He is giving a course that focuses on machine learning. In this course, Mike wants to teach to his students how sensor data is processed with machine learning algorithms. For this he needs raw data from multiple sources such as a smartphone and an environmental sensor network. With help of ManySense, Mike and his students can focus entirely on sensor data processing without spending much time on data collection details.

4. System Implementation

In the following sections, we describe the implementation details of ManySense including its subscription mechanism, raw data collection, context inferencing, Adapter SDK, and exception handling.

4.1. Subscription. Third party applications wishing to subscribe to raw sensor data or inferred context data must implement the SensorEventListener (Listing 1) or the ContextEventListener interface, respectively, and add themselves as subscribers to ManySense through the SubscriberConnection class. The event listener interfaces provide a mechanism to notify the subscribers when data is updated, an error occurs, or ManySense service binding changes.

Subscribing applications must specify a data retrieval request upon subscription. The request is expressed either by a query object or by a string containing a query written in ManySense query language. A query specifies the type of data, the commands that should be executed on start-up, the filters to be used on raw data, and a schedule for the subscription. Because ManySense is running in a separate process, the SubscriberConnection class sends the request query to ManySense by interprocess communication (IPC).

After implementing a listener interface and determining a data retrieval request, the subscriber binds to the ManySense service with the SubscriberConnection class that handles the complex binding process and acts as an interface between ManySense and the subscriber. Communication between the SubscriberConnection and ManySense is established using the Android interface definition language (AIDL) that enables efficient IPC on Android. The advantage of this approach is increased performance due to the usage of multiple threads instead of a single thread used by the Messenger, an alternative Android IPC method. In transmitting objects over IPC, we chose to marshal them using Parcelable interface instead of Serializable because the latter was a major performance bottleneck in our experiments. Before using AIDL we experimented with the Messenger because its single thread approach is less complex. Performance differences of AIDL and Messenger approaches are discussed later.

Listing 2 shows code snippets that connect the subscriber application with ManySense. SubscriberConnection object is first declared in a place that is relevant to the desired...
private SubscriberConnection conn;

// Initialize connection
conn = new SubscriberConnection(this);
conn.registerListener(listener);
conn.openConnection();

// Subscription code
// Unsubscribing
conn.sendQuery("unsubscribe from ∗");
conn.closeConnection();

Listing 2: Subscribing to ManySense with SubscriberConnection.

subscribe to phone/accelerometer, smartwatch/accelerometer commands delay 50 filters lowpass 0.6 modify subscription in phone/accelerometer commands send only new data true send until 22-10-2014 12:00:00 unsubscribe from ∗ subscribe to context/activityGoogle commands delay 10000 send for 30 minutes

Listing 3: Examples of MSQL queries.

lifecycle of the connection. The SubscriberConnection is instantiated with the current context so that it can bind to the SubscriberService running in ManySense. Then the listener needs to be registered. Calling the openConnection() binds the subscriber application to the service and closeConnection() unbinds it. To unsubscribe, the subscriber sends a query defining which subscriptions should be removed. Alternatively, the subscriber can remove all subscriptions, as shown in Listing 2. Closing the connection also removes all subscriptions. An example of subscription code can be seen in Listing 3.

When a subscriber application makes a request to the SubscriberConnection, it is forwarded to ManySense through IPC. Figure 4 illustrates this communication. Requests are received by the SubscriberService which parses MSQL queries into query objects and forwards the query objects to the SubscriptionHandler which stores information about all subscriptions. The SubscriptionHandler determines if the query targets context data or raw data and forwards it to the appropriate aggregator.

4.1.1. ManySense Query Language. In order to facilitate developers’ interaction with ManySense, we developed the ManySense query language (MSQL) which can be used for creating, modifying, and removing subscriptions in ManySense. We defined the language and generated a parser using ANTLR, a tool for defining structured languages. Listing 3 shows representative examples of MSQL queries. The first query subscribes to raw data from the phone’s built-in accelerometer and a smartwatch’s accelerometer. It also specifies that the data should arrive at 50 millisecond intervals with low-pass filter applied. The second query modifies the first subscription by specifying that only new data should be sent and that the subscription should automatically end at the given time.

In all query types it is possible to define the types of data sources and data type that the query will target. In the case of subscription and modification queries it is mandatory to specify at least one pair of data source and data type. In subscription removal queries it is possible to use an asterisk as wildcard for subscribed data sources. In subscription and modification queries it is also possible to define commands, filters, and ending constraints. Multiple commands can be given with or without parameters. Multiple filters can also be
applied to the data. Ending constraints define scheduling for the subscription. It is possible to specify that the subscription continues indefinitely (default), until a given time, for a given time period, or until a given amount of updates have been sent. It is also possible to send multiple queries at a time, separated by a semicolon.

4.2. Raw Data Collection. The raw data collection component is responsible for handling requests for raw data. The raw data aggregator (RDA) receives requests and tries to fulfill them as the best it can. It requests an updater to be started once it finds a suitable sensor. Updaters are responsible for keeping the subscribers updated. They poll the given sensor for new data at set intervals. There is an updater running per every subscriber's every subscribed sensor. This is because all the subscribers may have different parameters for how they wish to receive data. The sensors in turn are represented by the SensorAdapter interface. This allows ManySense to handle heterogeneous sensors through a common interface.

The adapter design pattern is used when integration of two incompatible interfaces is desired [32]. The main idea is that an adapter is provided to mediate communication between interfaces by appropriate conversion. We use the adapter pattern to provide a common interface for all sensor devices. The SensorAdapter is an interface that is implemented by adapters for each sensor device providing raw data. It defines the general capabilities that all adapters must have. These general capabilities include initializing, starting, stopping, and deinitializing the sensor, as well as getting the latest event and type of the sensor. All of these adapters are held and mapped by their sensors’ types in respective device objects. A device object contains information about the device’s physical location on the user's body such as on chest, wrist, or hip. This information can be used by activity detection algorithms, for example. The device objects are held in the RDA which maps them by the device type. This categorization makes it possible to find a requested sensor with just the device type and sensor type. For example, if the subscriber requests data from a smartwatch, ManySense may retrieve data from a Sony, Pebble, Samsung, or any other smartwatch that the user might have and that supports the requested sensor type.

The RDA gathers sensor data through events sent by individual Sensor Adapters. The data is cached locally in SQLite database and is then sent to the subscribers. It may also be sent to a remote server for long-term storage and analysis.

The communication between adapters and the physical sensors depends on the sensors. For internal sensors of the smartphone, communication is trivial; the adapter simply adds itself as a listener for updates from the Android sensor API. As for Bluetooth devices, the process is more complicated. First the adapter needs to see if the device/sensor exists and then it has to connect to the device. After the connection is made, the rest of the process continues on a new thread. The new thread then needs to get data from the device, parse it, and send events to the RDA as data is collected. This approach was used in implementing the adapter for the Zephyr heart rate monitor. In the case of adapters for online data sources such as weather data services, HTTP connection running in a separate thread must be used. The complexity of such connection depends on the data source API.

There are two basic requirements for potential sensor devices to be used with ManySense. First, the communication protocol they use should be open to make access to sensor data possible. With closed protocols creating an adapter would become tedious as it would require some form of reverse engineering. Second, the smartphone needs to be able to communicate with the sensor device. This can be typically done by Bluetooth, USB, WiFi, or mobile networks. For example, accessing a ZigBee wireless sensor network can be done by a USB or Bluetooth connection to a sink node. The devices that were used in developing and testing the middleware were Samsung Galaxy S3 smartphone, Zephyr Bluetooth heart rate monitor, and Sony SmartWatch SW2. The sensors that are available in each device are shown in Table 1. Virtual sensors (e.g., orientation, linear acceleration) provided by the Android API have been omitted from the table. These devices were chosen because their protocols are open, documentation is abundant, and they are Android compatible.

At the moment of writing this paper, we have completed adapters for all of the sensors mentioned in Table 1 and also an adapter for getting temperature data for the user's current location by using the online OpenWeatherMap API.

4.3. Context Inferencing. The context inferencing module is responsible for providing inferred context data to the subscribers. The context data aggregator (CDA) selects the appropriate context inferencing adapter (CIA) to fulfill a request. Context data events are sent through the same SubscriptionHandler interface as the raw data events. The event argument is given as a character string, thus allowing the context data to be presented in JSON, XML, or similar structured format.

The context inferencing module follows the raw data collection module in its architectural design. Adapters take care of requesting necessary input data from the RDA and executing inference algorithms on raw data. For example, an activity detection adapter might use accelerometer data from a smartphone and from a smartwatch to infer the user's activity. Currently we have two CIAs implemented as proofs-of-concept: (1) an activity detector based on Google's Activity Recognition API and (2) a step counter which uses accelerometer data to count the user’s steps.

4.4. Adapter SDK. The Adapter SDK provides tools for making new Sensor Adapters and CIAs. It includes simulator applications for both adapter types, along with documentation and example adapters. The purpose of the simulators is to run a new adapter through example scenarios and see that it matches the requirements set on them. Figure 5 shows the ManySense adapter simulation tool (MAST) after a successful test run of a weather data Sensor Adapter. The simulator checks that the adapter's preferences are valid and that it is capable of producing data at a consistent rate. In the figure
you can see also that the rate is changed around midway to confirm that the adapter responds to this command. In the end the simulator also checks that all invalid state transitions throw exceptions properly and that all valid transitions do not.

Many of the software components on which the simulators are based are taken from ManySense. The RDA and the CDA are the only customized components. The simulator is available as an application project where the developer must add new adapter for testing.

All Sensor Adapters should extend the AbstractSensorAdapter as it implements the SensorAdapter interface and provides the basic lifecycle management for all Sensor Adapters, thus reducing development work. The developer can then concentrate on the essentials such as defining how data is acquired, how it is processed, and how subscriber commands are handled. Extending this class makes sure that the Sensor Adapter passes the state transition tests.

4.5. Exception Handling. Exception handling is needed for fault tolerance and recovery. If an error occurs, ManySense will notify the sensor’s subscribers with a message that carries the exception that occurred. For example, if the subscriber sends an invalid command to a sensor, the exception that occurs is returned to the subscriber. This is important because it does not cause ManySense to crash and the subscriber can decide what to do with the exception.

If a subscriber forgets to unsubscribe and ManySense tries to send the subscriber a message, a DeadObjectException is thrown by Android. This exception is caught by ManySense and the subscriber is unsubscribed from everything. If there are no more subscribers left, ManySense will shut down to save resources.

Another typical erroneous situation that may occur at runtime is that the Bluetooth connection to a device is lost. If this happens, the exception is forwarded to the subscriber and the adapter deinitializes. Then it is up to the subscriber to choose what to do. It could, for example, try to subscribe to the sensor again in a moment or give a prompt to the user to make sure the connection is possible and provide a button for retrying the connection.

5. Evaluation

ManySense BSN middleware was evaluated and analyzed from three perspectives. First, we evaluated the performance of ManySense by comparing its capability of sending data at short intervals with multiple subscribers against direct listening of sensors. Because ManySense can be used by many subscribers at the same time, it is important that it is able to perform well under load. Performance was also tested by profiling CPU, power, and memory usage of multiple services using ManySense simultaneously and comparing those data to services directly using the same data sources. Second, as extensibility was one of the design goals, we evaluated extensibility of the adapter interfaces by creating a new Sensor Adapter and analyzing the process from the developer’s perspective. Third, accessibility as another design goal was evaluated by coupling ManySense with an existing exergame.

5.1. Performance. The device used for performance evaluation was Google Nexus 7 tablet running Android version 4.4.3. The device was practically brand new and it had only the default applications installed, thus minimizing other applications’ influence on the results. Nexus 7 was chosen because it provided us with the capability of measuring power consumption of applications using Qualcomm’s Trepnn profiler that gathers data on the usage of CPU, memory, and power.

5.1.1. Load Performance. Load performance test was conducted by creating a predefined number of objects of a nonstatic nested class in an Android Activity. The purpose of these objects is to simulate multiple subscribers accessing ManySense concurrently. Each subscriber object bound to the service and requested the phone’s accelerometer data at a predefined rate. This test was conducted with two versions of ManySense using different interprocess communication (IPC) methods. The first version used the Messenger class, whereas the second version used an AIDL interface. After 10 seconds had passed, the subscribers were stopped and each of them calculated the average delay. Then the average of the
subscribers’ averages was calculated and plotted on a chart. We also measured load performance when directly accessing the Android Sensor API without ManySense.

Figure 6 illustrates the results of the load performance test while using the Messenger class for communication, and Figure 7 shows the results while using an AIDL interface. Subscribers (listeners) using ManySense are plotted against subscribers using the Android Sensor API without ManySense (No MW). Each line indicates a configured update delay in milliseconds. A completely flat horizontal line is optimal because it corresponds to requested delay. There is a big difference in performance between the Messenger and AIDL versions. In the AIDL test, we can see only a slight difference with directly using the Android Sensor API when the subscriber count was raised to 100 and the requested delay was set to 0 milliseconds. The Android Sensor API provided the requested data at 10 millisecond interval, whereas ManySense produced a 16 millisecond average interval. The reason for the slight increase is that all of the subscribers of ManySense have a dedicated Updater that makes sure they have the newest data. When all of the 100 Updaters send events nearly some of the events may simultaneously be lost due to the buffer becoming full.

During load performance tests we found that having many subscribers raises problems relating to multithreading. ConcurrentAccessExceptions emerged because threads were trying to access the data in ManySense at the same time. Synchronizing the methods or the areas of code that cannot run at the same time solved the problem, but synchronization may have a negative effect on performance. However, as the results presented above were acquired with synchronization, we can assume that possible performance drop caused by synchronization is minimal.

5.1.2. CPU, Memory, Power Usage, and Code Complexity. For this test we created four background services in a testing application. The first two gathered data from the built-in accelerometer at 50 millisecond interval and from the OpenWeatherMap API at 500 millisecond interval, respectively. The other two services collected the same data using ManySense.

The Trepn profiler application for measuring the usage of CPU, memory, and power was set to profile at 500 ms intervals. The test device had Google account synchronization off and all other applications closed. The screen timeout was set to 30 minutes. At the start of profiling, a baseline for measurements was captured for 15 seconds. Profiling was done for a period of 20 minutes. The data was exported to a CSV file and imported into Excel which was used for analysis and visualization. This data included performance data for the whole system as well as data concerning the testing application and ManySense.

Figure 8 compares the system’s total power consumption between the two test runs. The average was 1190.6 mW with ManySense and 1191.1 mW without. Figure 9 compares the application-specific CPU usage and shows a small increase when ManySense was used. The average was 3.66% with ManySense and 2.21% without. One peculiarity in the figure is the spikes in the power consumption that occur at regular intervals. Since they occur in both graphs, they are most likely caused by the Android system. The profiler also measured network usage for other applications during the test. For example, Google Play Services and Google Search sent and received data at times corresponding to some spikes. These spikes did not have a major effect on the results because they appeared seldom. To compare memory usage we used the average measured resident set size between the two test runs, as shown in Table 2. The resident set size is the amount of RAM the application is using. In the case of ManySense, the first amount is what was used by the testing application and the second one is what was used by ManySense.

Although the test services were programmed with compactness, there was a big difference in their code complexities. We counted the amount of lines of code in each service excluding empty lines, comments, import statements, and
package declarations. There were 18 lines of code that were mandatory and common in all classes.

Including the mandatory lines, both services using ManySense used 39 lines of code. The service using the Android Sensor API directly used 32 lines of code, and the service using OpenWeatherMap API directly used 85 lines of code.

5.2. Extensibility. We evaluated ManySense’s extensibility by creating a new adapter for the phone’s magnetic field sensor using the Adapter SDK. First, a class was created that extended AbstractSensorAdapter and implemented its abstract methods. For phone sensors this is relatively simple to do due to Android’s Sensor API. Then in the RDA’s constructor we added the lines necessary to make it aware of the new sensor type. To test the adapter, we requested for its data in a subscriber application. The whole process took around 15 minutes. If the procedure forgetting data from the sensor is more complex, as is the case with Bluetooth devices, programming an adapter will take some time longer.

Listing 4 shows a code snippet from the implemented magnetic field adapter. The onIniatilize method is called when data is requested from the adapter, but it is not yet running. It first gets a reference to the Android Sensor API’s SensorManager class for accessing in-device sensors.
Magnetic field sensor object is then acquired through the SensorManager object. The onSensorChanged method is called when the phone sensor sends an update to the adapter. When this happens the update is inserted into a ManySenseEvent object and set as the latest event which an Updater can pick up.

To further test the extensibility of ManySense, we asked 24 undergraduate computer science (21) and digital media (3) students attending a context-aware application development course to make Sensor Adapters for ManySense and test them using the simulator. The students were given the Adapter SDK which contained a skeleton of the adapter to fill in the implementation details. Their task was to implement a Sensor Adapter for retrieving temperature data from the OpenWeatherMap API. To evaluate the difficulty of the task to the students, we analyzed their weekly learning diaries where they discuss their experiences and learning process. Sample quotes are as follows:

“I thought it was so simple to use” (4th year student, female),

“It was easy to complete project” (4th year student, male),

“I couldn’t understand well why I use it. It was difficult.” (4th year student, female).

In general, the task was easy for them, though some of them had trouble understanding how the adapter connects to the rest of the system. As we can see, for some of the students understanding how the adapters work was easy, but not for everyone. This experiment showed that while developing adapters with examples and template code is easy, the developers need to understand the whole picture to make the development easier.

5.3. Accessibility. Accessibility of ManySense was evaluated by integrating it into an existing application. For this we used Calory Battle AR [33], an Android-based mobile augmented reality exergame that aims to promote physical activity among children. Unlike stationary living room exergame systems such as Nintendo Wii, Calory Battle AR's game play is tied to the real world context and is designed to be played outside. No special hardware is needed; all the user need is his/her smartphone and the printed image targets.

The player's role is to find target locations guided by a GPS map and perform whatever tasks given there. For example, the player might need to defuse an augmented reality bomb or answer a quiz. Optional time limit encourages the player to run. The platform was made extensible so new kinds of tasks can be added easily. For this evaluation we created a new task which requires the player to spin around N times. The task requests orientation sensor data from ManySense at the highest speed possible.

Integrating ManySense into Calory Battle AR started with including a JAR library file containing necessary class files. Then the permission to use ManySense was added to the application's manifest. Next we added an object of SubscriberConnection to the new task's activity and registered it as a subscriber (Listing 2). After implementing all SensorEventListener interface methods (Listing 1), we made the task to subscribe to orientation data when the service was bound in the serviceBindingChanged method. Finally, upon receiving data in the sensorUpdated method we applied a spin detection algorithm. Given that the spin detection algorithm was already prepared, the process of integrating ManySense into Calory Battle AR took around 30 minutes.

6. Discussion

ManySense was created to allow the developers of third party applications to access heterogeneous sensor data and inferred context data in a uniform manner. The evaluation results indicate that ManySense performs well under load and it is fairly easy to create new adapters as well as to integrate ManySense into an existing application. As we saw in the load performance tests, an application can receive updates at optimal speeds with and without ManySense, but in the latter case the development effort is higher and is likely to result in duplicate code written by different developers.

Other performance tests displayed that using ManySense causes a slight increase in CPU and RAM usages. The difference in code complexities of the test services was more obvious. The service utilizing the built-in accelerometer directly was the simplest one, and thus we can conclude that if the application only needs a single source of data from the Android Sensor API, it is simpler to do it without ManySense. However, if more data sources are needed or getting the data...
requires multithreading, creating connections and parsing data, then it is much simpler to use ManySense. It should be noted that instead of using two services to test ManySense performance, we could have made only one service that would have subscribed to both accelerometer and temperature data by modifying the query. Even if different parameters were added, such as scheduling or filtering, it still would not increase the complexity of the code.

The biggest theoretical bottleneck of ManySense is the IPC which is subjected to a 1MB communication buffer. If the buffer becomes full, the sent data is lost. At the moment we do not see that this is becoming a practical problem because in a real situation over 10 listeners would be unnecessary and our load performance test did not show any signs of decreased load performance. On Android only one activity is active (visible) at any given time, so as long as the subscribers only keep the connections open when they are active, the IPC bottleneck will not become a problem unless subscribers subscribe to a large number of adapters. A solution to the IPC bottleneck problem is to package ManySense into a library that developers can embed in their applications. Then there is neither buffer nor IPC to slow it down. The shortcoming of this approach is increased resource usage if there are multiple ManySense-driven applications running simultaneously. Thus, in the future we plan to have two versions of ManySense: one that is situated in a separate process and another one that is embedded in a third party application as library. This would allow the developers to choose the version appropriate for their needs.

While the results of this study are encouraging, they are merely an initial survey to ManySense's extensibility. As more Sensor Adapters will be developed, we could see applications accessing not only wearable Bluetooth-based sensor devices but also ZigBee and HTTP sources. Similarly, more sophisticated Context Inferencing Adapters can be plugged into ManySense. An increase in software functionality has a tendency of decreasing performance; thus, in the future we must focus on performance while keeping the system dynamic.

Various well-known software design patterns were used in ManySense development (observer, singleton, adapter, template method, and façade). These patterns provide solutions to recurring problems which help us avoid reinventing the wheel [32]. The observer pattern was used to decouple the adapters from the subscribers. The singleton pattern was used to make various components, such as the aggregators and the subscriber handler accessible from anywhere, and also to ensure that there only exists one instance of them at any given time. The adapter pattern was used with Sensor Adapters and Context Inferencing Adapters to provide common interfaces for the data providers. The template method pattern was used in the AbstractSensorAdapter to allow the individual adapters fill in their algorithm implementations while keeping the common code in a single place. Finally, the façade pattern was applied to the SubscriptionHandler to offer a simplified interface for the SubscriberService to the aggregators. By using these patterns we conformed to the design principles presented in Section 3 and made ManySense extensible and accessible while keeping the implementation decoupled and encapsulated.

ManySense was designed to be flexible in terms of adding support for new sensors and context inferencing algorithms as well as for enabling access for third party applications. However, there remain challenges that should be addressed in future research and development. Firstly, even though creating new Sensor Adapters and CIAs is easy, there is currently no way of adding adapters to ManySense at runtime. New adapters must be added to ManySense codebase which is then compiled and distributed to the end users. Android’s automatic software update option diminishes this challenge, but it still requires an action from the end user. A component-based framework such as Apache Felix (OSGi) could be used to overcome this challenge because it allows the deployment of new components at runtime, but performance of Felix on Android should be investigated first. Secondly, many consumer sensor devices use closed or incompatible protocols. For example, Garmin Forerunner sport watches use the ANT+ protocol which is not supported by most Android smartphones. Thus, adapter developers are limited to use products that are open and accessible. Finally, smartphone and sensor device resource constraints (e.g., CPU, memory, and battery) have been widely researched by scholars, but these have not been thoroughly considered in ManySense. More research is needed to ensure optimal utilization of scarce resources.

7. Conclusions

ManySense BSN middleware supports heterogeneous off-the-shelf consumer-oriented wearable sensors as well as other data sources that can be used by developers to create context-aware applications. ManySense also provides access to inferred context data such as the user’s current activity. This type of middleware is needed because most previous BSN middlewares were found to be focused on specific domains such as healthcare and many lacked flexibility in terms of supporting new sensor and context data sources. Furthermore, many previous systems do not provide access to data for third party developers. To overcome the shortcomings of previous systems, in ManySense design we emphasized extensibility and accessibility while minimizing dependency on external components. These and other design principles resulted in a system architecture which allows low-effort addition of new sensors and context inferencing algorithms through adapter interfaces. It also allows convenient subscription by third party applications that can not only acquire data from multiple heterogeneous sensors and context inferencing algorithms but also send queries to control them. The ability of ManySense to clean sensor data by optional filtering further reduces the development effort of third party applications.

The primary motivation for ManySense is wearable sensors in consumer applications, but ManySense can efficiently support any sensors as long as they can communicate with the mobile device. For example, we could write a Sensor Adapter that connects to a sink node of an environmental ZigBee-based sensor network over USB or Bluetooth. Similarly,
we could develop adapters that would link Bluetooth-based personal health devices (PHDs) to ManySense for healthcare applications. Additionally, any Internet-based data source can be connected to ManySense through the mobile device's Internet connectivity. We demonstrated this by creating an adapter to acquire weather data from the OpenWeatherMap API. Finally, the adapter interface could also be used to monitor the user's device usage patterns (e.g., frequently used applications and visited websites) in order to gain a better understanding of the user's operational context. Needless to say, this, as well as other user data collection, creates serious privacy issues that must be dealt with.

The context inferring module is currently a proof-of-concept of ManySense's extensibility and as such it does not yet provide sophisticated context inferring algorithms. Thus, our future research will focus on creating a CIA for detecting the user's activity reliably in real time. This would involve combining multiple sources of data and then we can improve the accuracy of existing algorithms that focus on a single data source such as the smartphone's accelerometer. For example, by using accelerometers of both smartwatch worn on wrist and phone inserted in pocket we could more reliably tell whether the user is running or rowing.

Thus far we have merely taken the first steps towards enabling easy construction of fully body- and context-aware applications. As we pointed out in discussion, there are several challenges to be tackled in future research. The next steps are to create more adapters for ManySense and allow the adapters to be installed at runtime. This would increase the flexibility and accessibility of ManySense.

**Conflict of Interests**

The authors declare that there is no conflict of interests regarding the publication of this paper.

**Acknowledgment**

This work was supported by the new faculty research fund of Ajou University (2012).

**References**

[1] M. Weiser, "The computer for the 21st century," SIGMOBILE Mobile Computing and Communications Review, vol. 3, no. 3, pp. 3–11, 1999.

[2] Y. Hao and R. Foster, "Wireless body sensor networks for health-monitoring applications," Physiological Measurement, vol. 29, no. 11, article R27, 2008.

[3] D. Malan, F. J. Thaddeus, M. Welsh, and S. Moulton, "Code-Blue: an Ad Hoc sensor network infrastructure for emergency medical care," in Proceedings of the Workshop on Applications of Mobile Embedded Systems (MobiSys ’04), pp. 12–14, ACM, 2004.

[4] G. Vironne, A. Wood, L. Selavo et al., “An assisted living oriented information system based on a residential wireless sensor network," in Proceedings of the 1st IEEE Transdisciplinary Conference on Distributed Diagnosis and Home Healthcare (D2H2 ’06), vol. 219 of BIBLIOGRAPHY, pp. 95–100, Arlington, Va, USA, April 2006.

[5] B. Kim, Y. Kim, I. Lee, and I. You, “Design and implementation of a ubiquitous ECG monitoring system using SIP and the zigbee network,” in Proceedings of the International Conference on Future Generation Communication and Networking (FGCN ’07), pp. 599–604, Jeju-Island, Republic of Korea, December 2007.

[6] S.-J. Oh and C.-W. Lee, "U-healthcare SensorGrid gateway for connecting wireless sensor network and grid network," in Proceedings of the 10th International Conference on Advanced Communication Technology (ICT ’08), vol. 1, pp. 827–831, Gangwon-do, Republic of Korea, February 2008.

[7] U. Mitra, B. A. Emken, S. Lee et al., “KNOWME: a case study in wireless body area sensor network design,” IEEE Communications Magazine, vol. 50, no. 5, pp. 116–125, 2012.

[8] K.-H. Park and J.-G. Pak, “An integrated gateway for various PHDs in U-healthcare environments,” Journal of Biomedicine and Biotechnology, vol. 2012, Article ID 954603, 7 pages, 2012.

[9] R. Harle and A. Hopper, “Sports sensing: an olympic challenge for computing,” Computer, vol. 45, no. 6, Article ID 6226522, pp. 98–101, 2012.

[10] D. K. Arvind and A. Bates, "The speckled golfer," in Proceedings of the 3rd ICST International Conference on Body Area Networks (BodyNets ’08), pp. 1–7, Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, Brussels, Belgium, 2008.

[11] A. F. Smeaton, D. Diamond, P. Kelly et al., “Aggregating multiple body sensors for analysis in sports,” in Proceedings of the 5th International Workshop on Wearable Micro and Nanosystems for Personalised Health (pHealth ’08), 2008.

[12] C. Seeger, A. Buchmann, and K. Van Laerhoven, “An event-based BSN middleware that supports seamless switching between sensor configurations," in Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium (IHI ’12), pp. 503–512, ACM, January 2012.

[13] K. Takeda, H. Erdogan, J. Hansen, and H. Abut, In-Vehicle Corpus and Signal Processing for Driver Behavior, Springer, New York, NY, USA, 2009.

[14] E. Miluzzo, N. D. Lane, K. Fodor et al., “Sensing meets mobile social network: the design, implementation and evaluation of the cenceme application,” in Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys ’08), pp. 337–350, ACM, New York, NY, USA, November 2008.

[15] M. Musolesi, E. Miluzzo, N. D. Lane, S. B. Eisenman, T. Choudhury, and A. T. Camp-bell, "The second life of a sensor—integrating real-world experience in virtual worlds using mobile phones," in Proceedings of the 5th Workshop on Embedded Sensor Networks (HotEmNets ’08), 2008.

[16] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, and A. T. Campbell, “The BikeNet mobile sensing system for cyclist experience mapping," in Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys ’07), pp. 87–101, New York, NY, USA, November 2007.

[17] C.-H. Wu, Y.-T. Chang, and Y.-C. Tseng, “Multi-screen cyber-physical video game: an integration with body-area inertial sensor net works," in Proceedings of the 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM ’10), pp. 832–834, March 2010.

[18] C. Garcia Wylie and P. Coulton, "Persuasive mobile health applications," in Electronic Healthcare, D. Weerasinghe, Ed., vol. 1 of Lecture Notes of the Institute for Computer Sciences, Social
Informatics and Telecommunications Engineering, chapter 11, pp. 90–97, Springer, Berlin, Germany, 2009.

[19] N. Crampton, K. Fox, H. Johnston, and A. Whitehead, “Dance, dance evolution: Accelerometer sensor networks as input to video games,” in Proceedings of the 6th IEEE International Workshop on Haptic, Audio and Visual Environments and Games (HAVE ’07), pp. 107–112, October 2007.

[20] K. Montgomery, C. Mundt, G. Thonier et al., “Lifeguard—a personal physiological monitor for extreme environments,” in Proceedings of the 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC ’04), pp. 2192–2195, September 2004.

[21] J. Zhong, M. Sinclair, and R. Bittner, “A phone-centered body sensor network platform: cost, energy efficiency & user interface,” in Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks (BSN ’06), Cambridge, Mass, USA, April 2006.

[22] H. Viswanathan, B. Chen, and D. Pompili, “Research challenges in computation, communication, and context awareness for ubiquitous healthcare,” IEEE Communications Magazine, vol. 50, no. 5, pp. 92–99, 2012.

[23] A. B. Waluyo, I. Pek, X. Chen, and W. S. Yeh, “Design and evaluation of lightweight middleware for personal wireless body area network,” Personal and Ubiquitous Computing, vol. 13, no. 7, pp. 509–525, 2009.

[24] J. Al-muhtadi, S. Chetan, R. Campbell, and M. D. Mickunas, “Middlewhere: a middleware for location awareness in ubiquitous computing applications,” in Proceedings of the 5th International Middleware Conference (Middleware’04), pp. 397–416, 2004.

[25] D. Carr, M. O’Grady, G. O’Hare, and R. Collier, “SIXTH: a middleware for supporting ubiquitous sensing in personal health monitoring,” in Wireless Mobile Communication and Healthcare, B. Godara and K. Nikita, Eds., vol. 61 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pp. 421–428, Springer, Berlin, Germany, 2013.

[26] A. M. Khan, A. Tufail, A. M. Khattak, and T. H. Laine, “Activity recognition on smart-phones via sensor-fusion and kda-based svms,” International Journal of Distributed Sensor Networks, vol. 2014, Article ID 503291, 14 pages, 2014.

[27] M. Baldauf, S. Dustdar, and F. Rosenberg, “A survey on context-aware systems,” International Journal of Ad Hoc and Ubiquitous Computing, vol. 2, no. 4, pp. 263–277, 2007.

[28] S. Nath, “ACE: exploiting correlation for energy-efficient and continuous context sensing,” in Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys ’12), pp. 29–42, ACM, New York, NY, USA, June 2012.

[29] O. D. Lara and M. A. Labrador, “A mobile platform for real-time human activity recognition,” in Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC ’12), pp. 667–671, January 2012.

[30] T. Gu, H. K. Pung, and D. Q. Zhang, “A service-oriented middleware for building context-aware services,” Journal of Network and Computer Applications, vol. 28, no. 1, pp. 1–18, 2005.

[31] C. Seeger, A. Buchmann, and K. van Laerhoven, “myHealthAssistant: a phone-based body sensor network that captures the wearer’s exercises throughout the day,” in Proceedings of the 6th International Conference on Body Area Networks (BodyNets ’11), pp. 1–7, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, Belgium, 2011.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vliss-Sides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, Boston, Mass, USA, 1994.

[33] J. Westlin and T. Laine, “Short paper: calory battle AR: an extensible mobile augmented reality exergame platform,” in Proceedings of the IEEE World Forum on Internet of Things (WF-IoT’14), pp. 171–172, March 2014.
Submit your manuscripts at http://www.hindawi.com