A Publish/Subscribe Middleware for Body and Ambient Sensor Networks that Mediates between Sensors and Applications

Christian Seeger
Databases and Distributed Systems
Technische Universität Darmstadt
Darmstadt, Germany
e-mail: cseeger@dvs.tu-darmstadt.de

Kristof Van Laerhoven
Embedded Sensing Systems
Technische Universität Darmstadt
Darmstadt, Germany
e-mail: kristof@ess.tu-darmstadt.de

Jens Sauer, Alejandro Buchmann
Databases and Distributed Systems
Technische Universität Darmstadt
Darmstadt, Germany
e-mail: buchmann@dvs.tu-darmstadt.de

***Abstract*—Continuing development of an increasing variety of sensors has led to a vast increase in sensor-based telemedicine solutions. A growing range of modular sensors, and the need of having several applications working with those sensors, has led to an equally extensive increase in efforts for system development. In this paper, we present an event-driven middleware for on-body and ambient sensor networks that allows multiple applications to define information types of their interest in a publish/subscribe manner. Incoming sensor data is hereby transformed into the desired data representation which lifts the burden of adapting the application with respect to the connected sensors off the developer's shoulders. Furthermore, an unsupervised on-the-fly reloading of transformation rules from a remote server allows the system's adaptation to future applications and sensors at run-time. Application-specific event channels provide tailor-made information retrieval as well as control over the dissemination of critical information. The system is evaluated based on an Android implementation, with transformation rules implemented as OSGi bundles that are retrieved from a remote web server. Evaluation shows a low impact of running the transformation rules on a phone and highlights the reduced energy consumption by having fewer sensors serving multiple applications. It also points out the behavior and limits of the application-specific event channels with respect to CPU utilization, delivery ratio, and memory usage.**

***Keywords*—wireless sensor network, body sensor network, middleware, publish subscribe systems, event-based systems, Android**

I. Introduction

Rapid development towards sensor networks that consist of on-body and ambient sensors have commenced a new paradigm in telemedicine and elderly care. With growing diversity of sensors the range of monitoring parameters and, therefore, the range of potential applications and solutions expands. By using the same sensor network for multiple applications, solutions can be adapted to individual requirements. Furthermore, in order to adapt the network to new sensing requirements and to replace malfunctioning sensors, changing sensor setups are indispensable. Having a variety of sensors and applications within one sensor network is promising but results in two major challenges:

1) How to deliver only the desired and permitted sensor readings to an application in order to ease application development, manage information dissemination, and save resources?

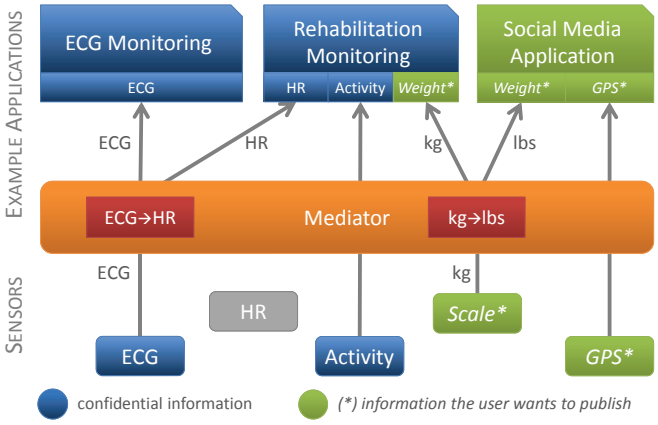


Fig. 1. Illustration of a mediator for patient monitoring that delivers only the desired and permitted readings and transforms readings from one data representation to another which obviates the need for some sensors (e.g., HR).

2) How to deal with the diversity of sensors and data representations without requiring application developers to adapt to new sensors and changing sensor constellations?

In order to emphasize the challenges, we will sketch a monitoring and rehabilitation application for a patient who suffered a heart attack as a motivational example. In the first weeks after the heart attack, the patient's heart is monitored with an ECG sensor. The delivered ECG stream allows a very accurate monitoring of the patient's heart for both, the cardiologist as well as a monitoring application running on a smart phone (cp. *ECG monitoring* in Fig. 1). Since rehabilitation programs are important for recovering and strengthening the heart after a heart attack, the patient is asked to perform light fitness exercises such as walking and cycling. Therefore, an additional *rehabilitation monitoring* application motivates and monitors the patient using activity, heart rate (HR), and body weight readings. In addition to the professional health care applications, the patient uses a private *social media application* for sharing the progress in losing weight. This application combines outdoor fatburning exercise information gathered from a GPS sensor with body weight readings.

The previous example consists of five sensors (ECG, HR, activity, GPS, scale) and three applications (*ECG monitoring,*

rehabilitation monitoring, social media application). This very simple example already maps the problem space. Each application is interested in specific sensor readings. Relaying all readings to all applications would require applications to filter incoming readings and, thus, consume additional resources. Furthermore, applications might have different confidentiality levels. The social media application that posts data to social networks should not get access to confidential sensor readings such as ECG streams, blood pressure and insulin intake. The ECG monitoring and the rehabilitation monitoring applications require both cardiovascular information. Since they operate on different data representations (ECG vs. HR), two different sensors are required although the ECG stream already contains heart rate information. In this paper, we introduce application-specific communication channels that deliver only desired and permitted sensor readings. Furthermore, we introduce a mechanism to convert sensor readings from one representation to another representation. Hence, a single ECG sensor can serve both a HR and an ECG application. This reduces the amount of sensors the patient has to carry and saves resources. In addition, replacing sensors does not require the developer to adapt her application to the new sensors. This is automatically done by the conversion mechanism running on the mediator.

Building on an event-based middleware architecture for on-body and ambient sensor networks which runs on a smart phone [1], we propose application-specific communication channels by making use of the publish/subscribe paradigm. Applications subscribe to event types of their interest and advertise event types they contribute. Before a subscription is permitted, a security manager checks the application's confidence level. Upon receiving an event, it is disseminated to subscribed and permitted applications. In order to avoid that developers must adapt their solutions to accommodate changing sensor configurations, we propose an adaptive event transformation mechanism that transforms the available information into the desired format. This could be a simple unit transformation from kilogram to pound, but also a more complex transformation from an ECG stream to a heart rate reading. A transformation that derives heart rate alarms based on heart rate and activity information and, hence, enriches the system's functionality, illustrates the capabilities of event transformations. Furthermore, a remote transformation repository allows the system to adapt to new requirements (e.g., sensors, applications) at run-time. For evaluation, we extended the Android implementation of **myHealthAssistant** [1]. Summarized, the two main contributions of this work are:

- *Publish/subscribe communication* for ambient and on-body sensor networks providing *application-specific communication* channels.
- An *adaptive event transformation* that converts information to the desired data representation and derives new events based on subscriptions. A priori unknown transformations are downloaded from a *remote repository* at run-time.

The paper is structured as follows: The next section describes the system architecture, especially the publish/subscribe messaging with application-specific communication channels and the approach of having an adaptive event transformation that mediates among publishers and subscribers. In Section III, we discuss the design decisions made for our prototype implemented as an Android application. We describe how to avoid encryption of the communication channels and how the system manages to change permissions at run-time. OSGi bundles allow installing, starting, and stopping event transformations without interruptions. The system's performance as well as the energy consumption for using only a single ECG sensor instead of ECG and HR sensors are analyzed in Section IV. Before conclusions and future work, Section V presents how our work compares to related projects.

II. System Architecture

This section describes the architecture of the proposed middleware for on-body and ambient sensor networks. The main focus lays hereby on the publish/subscribe messaging and the event transformation. The middleware, running on a smart phone, acts as a mediator among applications and sensors. It communicates with the sensors and allows applications to express their interest in specific event types. As soon as a new event becomes available, it is forwarded to the applications in the desired data representation. We decided for an event-driven middleware with publish/subscribe communication because: i) sensor constellations and running applications change over time, ii) most on-body and ambient sensors send their readings in an event-driven manner, and iii) sensors have no knowledge about the applications consuming their readings. Those characteristics fit the loosely-coupled publish/subscribe communication paradigm of event-based systems very well [2]. In [1], we proposed an event-based middleware that already handles the sensor communication, but lacks the support for application-specific communication channels and event transformations. Therefore, this paper describes how we extended the middleware in order to provide the missing functionality. After giving an overview of the system's architecture we will give a detailed description of the publish/subscribe communication and the adaptive context transformation.

A. Overview

Fig. 2 depicts the mediator's architecture consisting of several modules, communication channels, applications and the communication with on-body and ambient sensors. We will skip the description of the transformation manager on the bottom layer since Section II-C will describe it in detail. The **sensor modules** implement generic or sensor-specific protocols for communicating with individual on-body and ambient sensors. When starting, they advertise the event type(s) they will contribute to the system. Upon receiving a sensor reading, a corresponding event is injected to the system by sending it to the message handler. Some sensor modules provide a start/stop functionality to the message handler in order to stop the sensor if there is no subscription to the corresponding event type(s). This saves unnecessary sensor communication. The message handler in the intermediate layer manages advertisements, event subscriptions, and notifications. It will be described in Section II-B.

The top layer of Fig. 2 consists of a security manager, a system monitor and an event composer. The **security manager** provides information about the permissions of applications and modules. It is used to define which application is allowed to subscribe to which event type and to validate subscribers. By providing encryption keys, the security manager is used

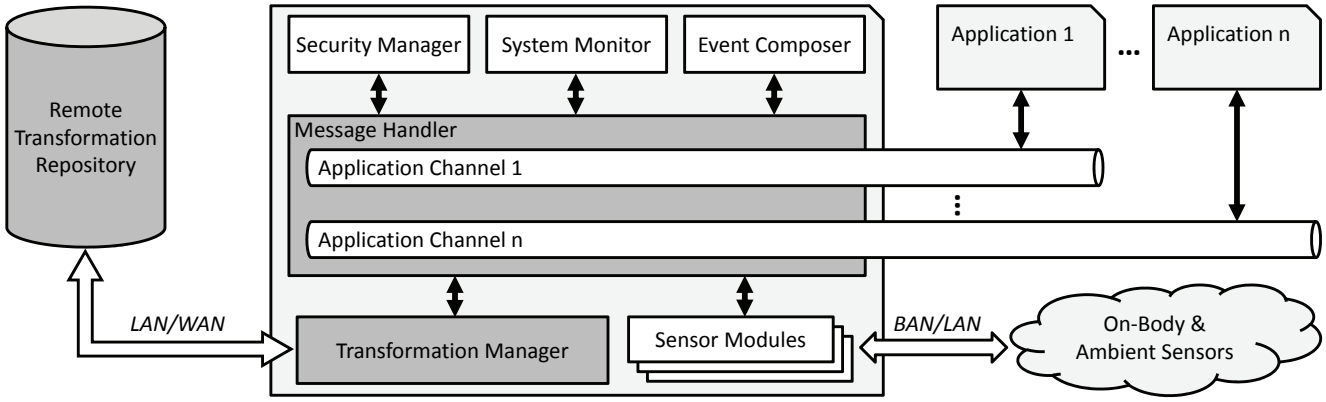


Fig. 2. Event-driven middleware architecture consisting of a `message handler` for managing application-specific communication channels based on subscriptions and a `transformation manager` for converting sensor data to the desired representation. New transformations are downloaded from the `remote transformation repository` at run-time. `Sensor modules` translate raw sensor data to sensor events and the `event composer` provides additional sensor fidelity information. The `security manager` is used for encryption, verification and managing permissions and the `system monitor` for keeping track of the overall system status.

to secure the communication channels between applications and the middleware. Since the middleware is designed for a smart phone, we can assume that applications and the security manager can easily connect to a public key infrastructure and download required encryption keys at run-time. The **system monitor** is responsible for monitoring the overall system. In case of a critical situation such as a low running battery, the system monitor injects an alarm event and starts appropriate reactions. This reaction is very system-dependent. In our Android implementation, for instance, the reactions to low battery power are a notification to the user as well as an SMS text message to a predefined phone number. In order to detect a crashed system monitor or a lacking connection to the network carrier which would hinder the detection and dissemination of further problems, heartbeat messages containing status information are periodically sent to a server. The **event composer** consumes sensor and application related events and provides extra functionality to the user and application developer. It identifies general situations on which the system has to react (e.g., critical vital signs) and creates a corresponding derived event. Furthermore, it detects inaccurate or invalid sensor data and emits events enriched with fidelity information as presented in [3].

B. Publish/Subscribe Messaging

Assuming a setup of various on-body and ambient sensors as well as applications using the sensor data, we extracted four main requirements on the messaging service:

- Seamless handling of changing sensor and application constellations
- Application-specific interest in certain event types
- Mechanisms for managing the data access and confidence levels of applications
- Mechanisms to secure the communication between applications and the middleware

In the following, we describe a messaging service that satisfies the requirements listed above. Communication between applications and the middleware is based on the publish/subscribe paradigm which decouples information producers from information consumers. Information is exchanged in form of events which consist of an *ID*, *event type*, *timestamp*, *producer ID*, its *payload*, and additional fields such as a *sensor type* and the *time of measurement*.The following communication aspects are handled by the **message handler**:

1) Communication channel: Each application connected to the message handler communicates over its own communication channel (cp. Fig. 2). In order to establish an individual channel, the message handler uses the security manager for checking the authenticity of an application and to handle a key exchange protocol if necessary. Having a secured communication channel between an application and the middleware prevents other applications from gathering classified information such as sensor readings with a higher confidence level and information about the connected sensors.

2) Advertisement: In order to announce future events to the system, applications and (sensor) modules send advertisement messages to the message handler. An advertisement consists of an *advertisement ID*, *event producer ID*, the *event type* that is advertised, start/stop capabilities of the event producer, and additional properties regarding the event producer (e.g., sampling rate). This information is stored and used for event subscriptions. If an event producer provides start/stop capabilities (e.g., an on-body sensor that can be controlled by the sensor module), the message handler decides whether to start or stop the sensor based on the event subscriptions. Event producers can revert their advertisements, in case they do not provide further events to the system which could be due to a disconnected sensor. If, as a result, this event type becomes unavailable to the system, subscribers are informed.

3) Subscription: In order to retrieve information from sensors as well as other applications and modules connected to the system, an application has to subscribe to event types. This is done by sending a subscription message to the message handler. Upon receiving a subscription, the message handler forwards the subscription's credentials to the security manager in order to verify the application's permission for subscribing to the requested event type. If the request is granted, the message handler checks whether the requested event type was

advertised. If not, a transformation request consisting of the requested event type as well as the advertised event types is sent to the transformation manager which in turn searches for a corresponding transformation (cp. Section II-C). If the subscription succeeds, an acknowledgment including the event producer's availability is sent and corresponding events are disseminated over the application-specific channel. The event subscriptions are done asynchronously, because requesting the transformation manager might take some time depending on the network connection and remote transformation repository. Granted but unsuccessful subscriptions (i.e. no producer available) are stored. As soon as an event producer for the requested event type becomes available, the subscribed application is informed and the event forwarding installed. Applications can unsubscribe from event types by sending a corresponding unsubscribe message.

4) Notification: As soon as an event (e.g., a sensor reading) becomes available, the message handler disseminates it to the subscribed consumers. Since events are disseminated over individual channels, they have to be sent multiple times which requires additional effort and could cause delays. On the other hand, the advantages of application-specific communication channels are less effort for filtering events on the application-side and security features which could become indispensable for health care applications.

In summary, the message handler processes the publish/subscribe messaging and keeps track of all event producers and consumers. It distributes events with respect to subscriptions and their permissions and seamlessly handles changes in sensor and application constellations. In case of a subscription without a fitting advertisement, the transformation manager is queried in order to find a matching transformation. The integration of a security manager and encrypted communication channels allow control over the data access and a secure communication between applications and the middleware.

C. Adaptive Event Transformation

Different applications have different requirements on data and its representations. The fitness application in our example operates on simple heart rate readings whereas the health care application requires ECG information. By transforming the ECG data into heart rate information instead of having an ECG and a heart rate sensor connected to the system, the overall energy consumption is reduced (cp. Section IV-B) and a more energy efficient system is achieved. In addition to this, a replaced or future sensor might provide the same sensor information in another representation (e.g., data format, unit, granularity). As a result, replacing a sensor would require application developers to adapt their applications to the new sensor. By introducing a layer between sensors and applications that transforms the available information into the desired format, the amount of potential applications and sensors is increased and future applications and sensors are supported. In this section we present the **transformation manager**, an adaptive event transformation approach that transforms data into the desired data representation and adapts its own behavior with respect to the actual system configuration at run-time.

1) Transformation Request: In case the message handler receives an event subscription to an event type which is not advertised, it sends an event transformation request to the transformation manager. Such a request consists of an *ID*, the *requested event type*, and a list of *advertised event types* which contains all event types that are currently advertised. Based on this information, the transformation manager checks whether there is a transformation from one or more advertised event types to the requested event type already stored or whether it needs to forward this request to a remote repository (cp. Fig. 3). In case of a successful local or remote request, the corresponding transformation is started and a notification message is sent.

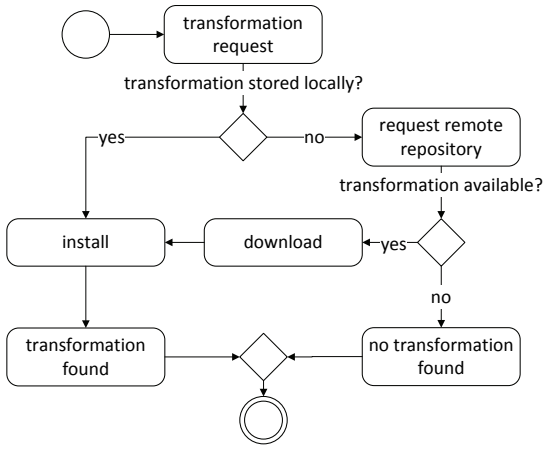


Fig. 3. Processing an event transformation request: If there is no transformation from the advertised event type(s) to the required event type stored locally, the request is forwarded to a remote transformation repository allowing the system to adapt to new requirements.

2) Transformation Life-cycle: As long as a transformation is not needed, it is stopped and does not require any CPU cycles and main memory. If a transformation is required, the transformation manager triggers the start of the corresponding transformation. When starting, a transformation subscribes to the event type(s) it requires for its transformation. In case of an untrusted remote repository, the subscriptions can be treated as an application subscription which requires a security check. After a successful subscription, the new event type is advertised including an obligatory start/stop functionality and the transformation starts working triggered by incoming events. If a transformation is stopped, it unsubscribes and reverts its advertisement.

Fig. 4 depicts the communication for starting and stopping a transformation. Communication with the remote repository is omitted for clarity. Case (a) shows the communication for starting a transformation: an application subscribes to the event type heart rate (HR). Since there are only event producers for ECG and blood pressure (BP) readings available, the message handler sends a transformation request to the transformation manager. Since the transformation manager has a transformation $T_{ECG \rightarrow HR}$ stored, it starts the transformation and sends an acknowledge message to the message handler which in turn informs the application. Furthermore, $T_{ECG \rightarrow HR}$ advertises events of type HR. In case (b), the application unsubscribes from HR events. Since it was the only application subscribed to HR events, the message handler sends a stop command to the transformation which has to support the start/stop functionality. The transformation stops, informs the transformation manager,

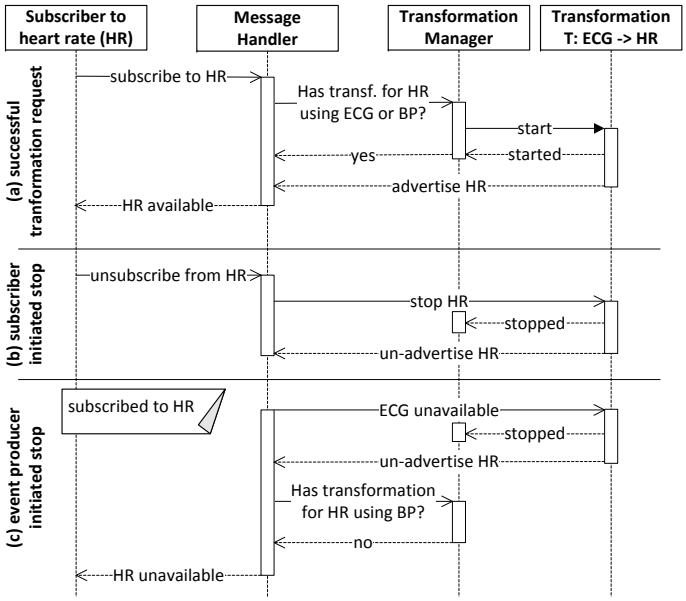


Fig. 4. Sequence diagram illustrating the communication among software components for starting and stopping an event transformation: (a) installing an event transformation for transforming events of type ECG to type HR, (b) the event transformation is stopped because the last subscriber to HR unsubscribed, (c) the event transformation is stopped because the event producer for events of type ECG is no longer available.

and reverts its advertisement of HR events. Case (c) depicts how the system reacts on a required event producer becoming unavailable. In this example, the ECG sensor becomes unavailable which causes a stop of the transformation $T_{ECG \rightarrow HR}$ and, hence, HR events become unavailable. Since there are still subscriptions to HR events, the message handler invokes a new transformation request containing only the advertised BP events. Because there are no matching transformations available, the request remains unsuccessful and, thus, the message handler informs the applications about the unavailable HR events.

3) Remote Repository: Every new event producer and event consumer potentially leads to the need for a new event transformation. With an increasing amount of applications and sensor devices available, the number of possible transformations increases rapidly. Having all potential transformations installed on a mediator for on-body and ambient sensor networks would overload most devices. Therefore, we decided for a **remote transformation repository** that enables the mediator to keep only transformations which are required and to download new transformations in case the requirements change. This way, resources are saved and an adaptive system is provided. If the device runs out of memory, transformations that were not activated for a while, are deleted. Having transformations running within the sensor network instead of a remote entity brings two main advantages: a) the system can operate without an Internet (or similar) connection, and b) sensitive user information remains within the sensor network.

For requesting a new transformation, the transformation manager connects to a remote repository. We assume that this is done via a secured network connection and after a successful authentication process in order to prevent the installation of malicious transformations. After the connection is established, the message handler's transformation request is encoded and sent to the remote repository. If the repository finds a match, it provides the corresponding transformation. If there are several potential transformations available, a cost function determines the best transformation for the requesting mediator. After the transformation is downloaded, the transformation manager has to provide an environment for executing loaded code for starting the transformation. In our implementation (cp. Section III-B), we decided for an OSGi framework.

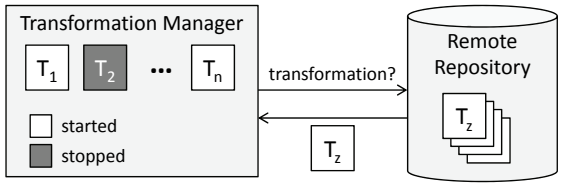


Fig. 5. Transformation manager with started and stopped transformations. A priori unknown transformations are downloaded from a remote repository.

Fig. 5 depicts the local transformation manager consisting of started and stopped transformations as well as a remote repository providing all potential transformations. Since transformations are allowed to consume multiple event types, they are not restricted to simple event type conversions. For instance in our implementation, we developed a transformation that detects a critical heart rate based on HR and activity events. In case an application requires such an alarm, this functionality can be added without user interaction or interrupting the system. The capability of reloading missing functionality enables an adaptive system behavior.

III. Implementation

This section discusses the implementation of the publish/subscribe messaging and the adaptive event transformation described in the previous section. For the mediator device hosting the middleware, we decided for a smart phone because it provides sufficient processing power, rich storage capacity and networking capabilities for on-body sensors, ambient sensors, and the Internet. The implementation extends a middleware for Android[1] devices which we proposed in [1]. This middleware already showed good results regarding battery lifetime, system performance, and stability [1], [4]. The following description and discussion is again divided in publish/subscribe messaging and adaptive event transformation. For a detailed description of other parts of the system, we refer to [1].

A. Publish/Subscribe Messaging

The message handler described in Section II-B is responsible for individual communication channels between each application and the middleware. In the Android operating system, there are three base mechanisms for inter-application communication that do not require user interaction [5]:

- *Services* expose their interface to other Android applications by using AIDL (Android Interface Definition Language). This allows applications to invoke methods from other applications and, thus, provides interprocess communication.

[1] http://developer.android.com

- *Content Providers* allow access to a structured set of data. Application can query this data set by addressing an application-defined URI.
- *Broadcast Receivers* receive Android `Intents` sent from any Android application. Those Intents can piggyback any data which implements the Android `Parcelable` object. Before sending an Intent to another application its content needs to be described. If the Android `IntentFilter` of a Broadcast Receiver matches to an Intent's description, it is received.

Since Android Services are built upon a remote procedure call (RPC) communication paradigm which has some drawbacks for publish/subscribe messaging [6] and Android Content Provider requires a query-based communication paradigm, they both do not fit very well to the event-driven sensor network communication. Android Broadcast Receivers support event-driven asynchronous communication across multiple channels and they allow piggybacking any data as long as it implements Android `Parcelable`. Therefore, we decided for Broadcast Receivers as the underlying communication mechanism. We distinguish between three ways of inter-application communication based on Android Broadcasts:

1) *Implicit Broadcast Intents* are received by any Broadcast Receiver having a matching `IntentFilter`.
2) *Signed Implicit Broadcast Intents* are only received by Broadcast Receivers that have a matching `IntentFilter` and a valid signature.
3) *Explicit Broadcast Intents* are only received by a specific Android destination class (application).

The authors of [5] have shown that signed Implicit Broadcasts and Explicit Broadcasts do not allow other applications to eavesdrop on the inter-application communication as long as the operating system is not penetrated. This saves the costs for encrypting the communication between applications and the middleware. A drawback of signed Broadcasts is the need of having the signatures already created beforehand and having them installed with the application which requires a re-installation of the system in case the security parameters change. Therefore, we decided for **Explicit Broadcasts** which saves the need of encrypting the communication and allows changing permissions at run-time.

In contrast to the architecture shown in Fig. 2, our Android implementation uses two unidirectional channels between the middleware and an application. Since other applications are not able to eavesdrop on Explicit Broadcasts, the message handler's receiver channel is the same for all applications. This simplifies the application development because the communication channel to the middleware is static and already known beforehand. Within a subscription request, applications transmit the component name to which the message handler is expected to send messages which again simplifies the development. Since the component name inherits the application's name, the security manager can easily check whether the request belongs to an installed and admitted application. An explicit authentication check is not implemented so far.

The message handler keeps track of all subscribed applications and advertised event types. In addition to this, it detects subscribers that are no longer listening on their channel and, hence, deletes their subscriptions and advertisements. Event types are structured in a tree and applications are allowed to subscribe to leafs as well as intermediate nodes. Upon receiving an event, the message handler looks up a routing table and disseminates the events according to the component names it has stored from the subscriptions. If event types become unavailable to the system, applications subscribed to this event type are informed.

B. Adaptive Event Transformation

For evaluating the adaptive event transformation, we implemented both the transformation manager running within the middleware and a remote repository providing a priori unknown transformations.

1) Transformation Manager: The transformation manager installs, starts and stops transformations with respect to the current system configuration. On an incoming transformation request, it first looks up the local transformation repository for a fitting transformation. If a transformation is found, it is started and subscribes to the event types required for the event transformation. The message handler treats transformations in the same way as applications. Upon receiving a stop or an un-advertise message from the message handler, the transformation stops, emits an un-advertise message and informs the transformation manager which in turn stops the whole transformation in order to release occupied memory.

In case a requested transformation is not stored locally, it has to be downloaded from a remote repository and installed without requiring a reboot or a user interaction. Usual Android installations require a restart of the application as well as user interaction which would hinder a seamless adaptation to new requirements. In order to still provide an automated adaptation to new requirements, we decided for making use of the OSGi (Open Services Gateway initiative)[2] framework. It is a Java-based framework that allows installing, updating, starting, stopping, and uninstalling application components (`bundles`) at run-time. Apache Felix[3] for Android provides OSGi support for Android applications and is used for this implementation. Consequently, transformations are implemented as OSGi bundles and the transformation manager makes use of the OSGi life-cycle.

2) Remote Repository: The remote repository is implemented as a web server. Transformation requests are sent as JSON encoded requests containing the requested event type and event types advertised to the requesting system. If a matching transformation is found, it is transmitted to the transformation manager, otherwise the system is informed that no transformation was found. In case of multiple matching transformations, the remote repository has to decide which one fits best. For this, a corresponding cost function based on the complexity of a transformation and the energy consumption for required event types could help determining the best transformation. By adding priority information to the advertised event types of a request, a requesting transformation manager could influence the decision making. Since the current implementation forwards the first transformation found, implementing a cost function remains as future work.

[2] http://www.osgi.org
[3] http://felix.apache.org/

Fig. 6. A Zephyr HxM Bluetooth heart rate sensor and a Corscience CorBELT Bluetooth 1-lead ECG sensor were used for testing the $T_{ECG \rightarrow HR}$ transformation and energy consumption. The HedgeHog sensor on the right provides acceleration data for the activity recognition needed in $T_{HRAlarm}$.

3) Transformation Complexity: Event transformations can have different complexity regarding processing power, memory usage and event subscriptions. We distinguish between three transformation types:

1) unit transformation, e.g.: $°C \leftrightarrow °F$, $kg \leftrightarrow lbs$, $mmol/L \leftrightarrow mg/dL$
2) type transformation, e.g.: ECG stream $\rightarrow$ heart rate (bpm), specific activity (e.g., standing, running, cycling, ...) $\rightarrow$ activity level (e.g., low, moderate, high)
3) reasoning: activity & HR $\rightarrow$ critical situation, accelerometer data $\rightarrow$ activity/step counter

Unit transformation are usually cheap to perform, whereas type transformations might become more complex. An ECG-stream-to-heart-rate transformation ($T_{ECG \rightarrow HR}$) for instance needs to detect the QRS complex in an ECG stream and calculate the heart rate based on the RR-intervals. Reasoning as the third type of event transformation might combine multiple events and use additional domain knowledge in order to derive an event of higher meaning. For instance, deriving a critical situation based on incoming heart rate and activity events ($T_{HRAlarm}$). In our evaluation, we implemented an event transformation of each type: $T_{kg \rightarrow lbs}$, $T_{ECG \rightarrow HR}$, and $T_{HRAlarm}$.

IV. Evaluation

The main task of the mediator is to disseminate events received from sensors, applications, and modules to other applications and modules. An evaluation on the original middleware [1], [4] has already shown that a system consisting of an accelerometer, a heart rate sensor, and several environmental sensors (e.g., blood pressure, scale, proximity) lasts at least for a day while performing activity recognition and health monitoring. In this paper, we will focus on our extensions, the performance analysis of the publish/subscribe messaging and the energy savings achieved by event transformations. For the performance analysis, we used HTC One V smart phones with a 1 GHz Tegra 2 single-core processor, 512 MB memory, and Android 4.0.3. All test results presented in this section are the average from three test runs à 100 seconds. For testing the transformations, we used a 1-lead ECG, a heart rate sensor, and an accelerometer which are shown in Fig. 6 and we conntected a Bluetooth-enabled scale.

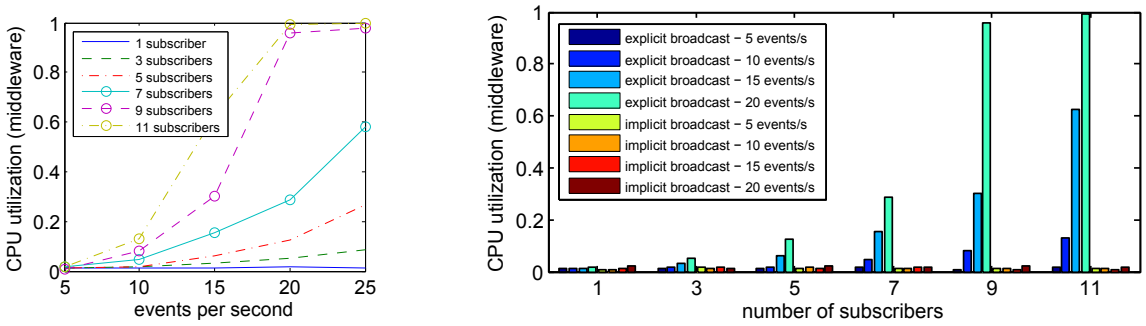
A. Publish/Subscribe Messaging

In order to evaluate the performance of the event notifications, we will analyze the system's behavior on an increasing amount of events per second as well as an increasing amount of subscribers. For this, we developed event generators that run in the same process as the middleware and operate as sensor modules. With 1 Hz frequency each event generator publishes events that consist of following fields: `ID`, `type`, `timestamp`, `producer ID`, `sensor type`, `time of measurement`, and an integer value as the `payload`.

Fig. 7(a), Fig. 8(a), and Fig. 9(a) depict the CPU utilization, delivery ratio, and memory usage for an increasing workload and up to 11 subscribed applications. For simplification, all applications are subscribed to the same event type which means that every event has to be resent by the amount of subscribers which can be considered as a worst case scenario. In a real deployment, the number of retransmission is usually lower. For only one application subscribed to the middleware, the CPU utilization stays below 1.4% and injected events are delivered to 100% for a workload of at least up to 25 events/s. If the number of subscribers increases, the performance starts dropping. For 3 subscribers and up to 15 events per second the CPU utilization is still low and the delivery ratio above 99%, but with an increasing workload the delivery ratio starts dropping. The same applies for an increasing number of subscribers. Further investigations have pointed out, that our system can handle up to 60 events/s. For instance, for 11 subscribers and a workload of 5 events/s, 5 events are published and $5 * 11 = 55$ events are delivered. For an increasing number of produced events per time unit, the maximum number of subscribers decreases. The memory usage increased with a growing workload. A current low-end smart phone possesses at least 512 MB main memory which makes up to 1.6% memory usage for 25 events/s and 11 subscribers.

In Section III-A we decided for Explicit Android Broadcasts in order to provide individual communication channels. In case all applications are subscribed to the same event type (as in the previous paragraph), Implicit Android Broadcasts that publish all events on a common channel would reduce the event dissemination effort. In order to analyze the overhead for providing application-specific communication channels, we ran the same tests with Implicit Broadcast messages. Figures 7(b), 8(b), and 9(b) depict the CPU utilization, delivery ratio and memory usage of both approaches with an increasing number of subscribers and up to 20 events/s. For Implicit Broadcast messages, the CPU utilization of our middleware is not affected by an increasing number of subscribers because each incoming event is forwarded only once. On the other hand, Explicit Broadcast messages save the message dispatching effort for the operating system since they are explicitly sent to Android components. Nevertheless, an almost 100% delivery ratio for dispatching Implicit Broadcasts compared to a decreasing delivery ratio for Explicit Broadcasts demonstrates the overhead for having application-specific communication channels. Furthermore, the additional publish/subscribe logic as well as advertisement and subscription lists cause a higher memory usage (cp. Fig. 9(b)).

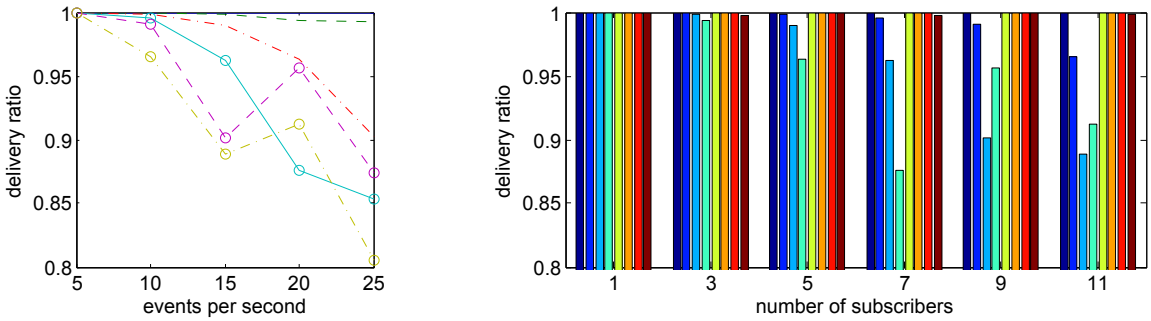
In summary, we believe that the advantage of secured and application-specific communication channels is worth the additional overhead. Furthermore, a message handling of up to 60 events per second is sufficient for most health care applications. Since most on-body and ambient sensors are battery powered, sending data is expensive and, thus, data is



(a) increasing workload

(b) comparison of Explicit versus Implicit Broadcast Intents

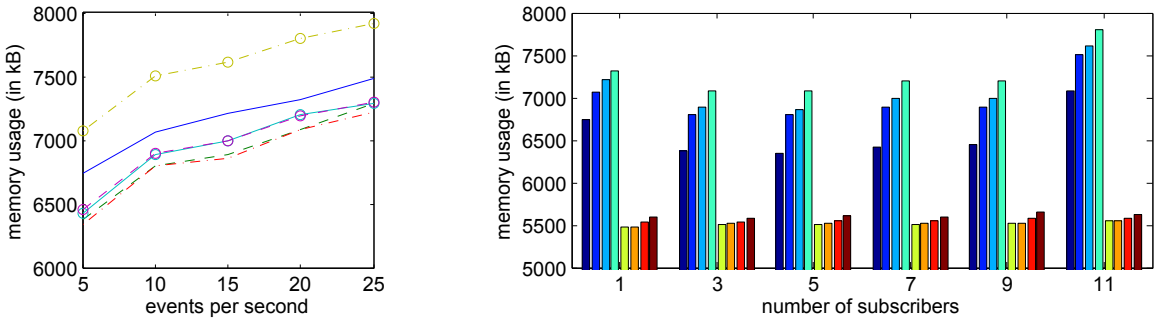
Fig. 7. CPU utilization of the middleware for (a) an increasing number of events/s, and (b) an increasing number of event subscribers.



(a) increasing workload

(b) comparison of Explicit versus Implicit Broadcast Intents

Fig. 8. Event delivery ratio for (a) an increasing number of events/s, and (b) an increasing number of event subscribers.



(a) increasing workload

(b) comparison of Explicit versus Implicit Broadcast Intents

Fig. 9. Memory usage of the middleware for (a) an increasing number of events/s, and (b) an increasing number of event subscribers.

aggregated and sent in a lower frequency. For example, the ECG sensor (cp. Fig. 6) we used in our deployment samples with 200 Hz but transmits with only 2 Hz frequency. The fitness diary application presented in [4] uses at most 6 sensors at the same time triggering one event per second. Therefore, even in a scenario in which the system consists of three accelerometers, a blood pressure and a heart rate sensor as well as a scale, the system still operates on the very left of our performance evaluation Figures 7(a), 8(a), and 9(a).

B. Adaptive Event Transformation

Referring back to the example described in the introduction in which a rehabilitation patient has two monitoring applications running on top of a body sensor network, we assumed that using a single ECG sensor and transforming an ECG stream to heart rate information requires less energy than using both sensor types. In order to validate our assumption, we implemented and installed a transformation from ECG to heart rate events and compared the energy consumption for different sensor constellations as shown in Fig. 10. The values for the energy consumption are taken from the PowerTutor application [7], [8]. Fig. 10 (a) shows the energy consumption in case of individual sensors for each application. In total, both the middleware and the Android system consume about 225 mW. Compared to this, the combination of an ECG sensor and a transformation that transforms ECG streams into heart rate events ($T_{ECG \rightarrow HR}$) consumes only about 191 mW and, thus, saves more than 15% of energy. In this case, the overall system lasts longer and does not require the user to wear an additional sensor. A comparison between (b) and (c) shows the additional energy overhead for performing the transformation which is

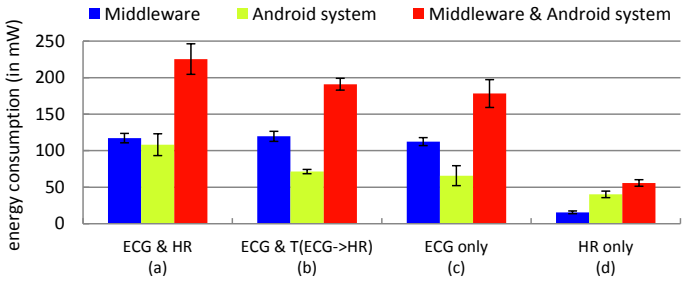


Fig. 10. Energy consumption for different sensor constellations. Having a transformation converting ECG streams to heart rate data (b) saves more than 15 % energy compared to having both sensors connected (a). Compared to an ECG sensor (c), a heart rate sensor (d) consumes perceptible less energy.

less than 7.5%. Fig. 10 (d) depicts the energy consumption for having only a heart rate sensor connected to the system. Since our heart rate sensor sends only 60 Bytes with 1 Hz frequency the energy consumption is much lower than the one for the ECG sensor sending 224 Bytes with 2 Hz frequency. The additional energy consumption produced by the two other transformations $T_{kg \rightarrow lbs}$ and $T_{HRAlarm}$ is only marginal.

As a remark, we have experienced that our ECG sensor is very prone to movement. As soon as the user was moving, the ECG curve started fluctuating with the result that our $T_{ECG \rightarrow HR}$ transformation was not able to detect QRS complex anymore and, thus, stopped delivering heart rate events.

V. Related Work

This section reviews related work in the area of middleware for (body) sensor networks that supports individual data delivery for multiple applications. For a detailed overview of applications and middleware approaches for body sensor networks and their health care applications, we refer to the surveys of [9], [10].

The authors of [11] propose a middleware for body sensor networks (BSNs) that is running on both, a mediator (e.g., mobile phone) as well as on sensor devices. Sensors advertise their capabilities to a service discovery daemon running on the mediator which allows applications to express their information needs and requirements. Based on the applications' requirements and the sensors' capabilities, sensor nodes are configured and sensor data is collected and delivered. Sensors can be added and removed at run-time. Compared to our work, this approach provides a more fine-grained specification of requirements such as delivery guarantees, sampling rate, and network bandwidth. In order to grant the middleware more control over the sensor nodes (e.g., adjust the sampling rate), specialized sensors that run the middleware are required.

In [12], [13], [14] LiteMWBAN, a middleware for medical BSNs that supports multiple sensors and applications, plug and play features, and resource management is introduced. Similar to the use case presented in our introduction, they consider ECG monitoring and a heart rate and activity monitor as two applications running on the same sensor network. Resource control messages allow checking and changing resource properties of sensor node which in turn requires the nodes to support the middleware's API.

The Self-Managed Cell (SMC) [15], [16] is a middleware which consists of a policy-based architecture that supports autonomic management and self-configuration for BSNs. A discovery service detects new sensor devices and removes subscriptions of disconnected subscribers. A content-based subscription mechanism allows the specification of filters to subscriptions such as "subscribe to heart rate values greater than 100". Policies define how the system should adapt in response to specific events which is similar to our transformations. Furthermore, authorization policies define permitted actions under certain circumstances. By providing content-based subscriptions, SMC allows application developers to define information of their interest very precisely.

The authors of [17] introduce MiLAN, a middleware to support multiple wireless sensor network (WSN) applications. Applications define their QoS requirements over time and how to meet these requirements using different sensor combinations. Based on different priority levels of applications and information about the available sensors and their status, MiLAN continuously adapts the network configuration to meet the applications' needs while maximizing the system's lifetime. For providing a proper management of the sensor network, MiLAN requires a tight integration with sensors and protocols.

Mires [18] is a publish/subscribe middleware for sensor networks. Similar to our approach, applications subscribe to event types that were advertised by event producers.In contrast to our work, Mires is a distributed middleware running on multiple sensor nodes that performs a message routing based on subscribed nodes. Furthermore, an aggregation service allows subscriptions to aggregated data which reduces the network load by performing in-network aggregation.

Unlike our work, the presented projects rely on sensor nodes that implement at least parts of the middleware. For WSNs, this is desired in order to have more control over the nodes and, thus, operate in an energy efficient way. In the area of BSNs, there are already a lot of off-the-shelf sensors from different manufacturers available. Our approach is to develop a system that utilizes these sensors and allows building a health monitoring system with already available sensors. A drawback of using unmodified sensors is the limited control. If future sensors provide more management functionality than just switching them on and off, they can advertise their extra functionality to the message handler and allow the middleware to have more control over the sensor nodes. The MobiHealth project presented in [19] also operates on off-the-shelf sensors while having main focus on the network infrastructure among BSNs and health care providers.

By introducing event transformations, we proposed a mechanism that converts sensor readings into the desired format. This could be a simple unit transformation, but also the derivation of a new event. One of the main objectives in [11] is to "*convert the data collected by the lower layer to relevant information in a human model*" which provides a similar functionality. The Self-Managed Cell [15], [16] introduces a policy service that allows the definition of event-condition-action (ECA) rules which can be used for event transformations. Both projects do not provide a mechanism to automatically reload missing transformations. In [20], a health care platform running on Android devices that provides loading OSGi bundles (i.e. additional functionality) at run-time

is presented. The focus of that work relies only on a dynamic software adaptation, but not on BSN applications.

VI. Conclusions

Many health care applications gain from the broader range of potential sensors provided by a mediator between sensors and applications. Introducing new sensors is done at a single point, the mediator, and does not require changing each individual application. Furthermore, the combination of multiple specialized applications running on the same set of sensors, provides a swift deployment process. The adaptation to new requirements is done by changing or adding only a specialized application instead of changing the whole deployment. Especially for health care scenarios, application-specific permissions to access sensor data are inevitable.

The contributions of this work are application-specific communication channels to a mediator for body and ambient sensor networks as well as an adaptive event transformation. Applications subscribe to desired event types and advertise provided events. After checking the validity of a subscription, approved applications are equipped with their own communication channel to the mediator and receive upcoming events on this channel. In case of a subscription to an event type that was not advertised, the system tries to find an event transformation from one or more of the advertised events to the desired event. A remote repository allows the adaptation to new requirements (i.e. applications, sensors) at run-time which makes adaptations of individual applications superfluous. In addition, sensors can be even replaced by event transformations which results in less wireless communication and, thus, energy savings.

An Android implementation of the system served for the system evaluation. Communication channels are established with explicit Android Broadcast Intents which saves the need for encryption and allows changing permissions at run-time. Event transformations in form of OSGi bundles are downloaded from a web server. The performance analysis on a low-end smart phone proofed the handling of multiple applications. It serves three applications with a rate of 15 events per second and per subscriber which is sufficient for most health care applications. We implemented three events transformations (i.e. $T_{kg \to lbs}$, $T_{ECG \to HR}$, $T_{HRAlarm}$) and showed the energy savings of replacing a heart rate sensor by the $T_{ECG \to HR}$ transformation.

For future work, we will implement more event transformations and analyze how the system scales for an increasing number of transformations running simultaneously. Furthermore, we like to apply the approach of a remote repository to sensor modules. Based on a sensor's identifier, the corresponding module is downloaded and an automated adaptation to new sensors and sensor protocols is provided. We also plan to introduce additional requirements in the event subscriptions in order to adjust the sensors depending on the applications' needs. For this, we will develop configurable accelerometers.

Acknowledgment

Gratefully supported by the German BMBF Software Campus (01IS12054) and the German Research Foundation (DFG) within the research training group 1362 Cooperative, Adaptive, and Responsive Monitoring in Mixed Mode Environments.

References

[1] 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 2012)*. ACM, December 2012.

[2] A. Hinze, K. Sachs, and A. Buchmann, "Event-based applications and enabling technologies," in *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS 2009)*, July 2009.

[3] C. Seeger, A. Buchmann, and K. Van Laerhoven, "Wireless sensor networks in the wild: Three practical issues after a middleware deployment," in *the 6th International Workshop on Middleware Tools, Services and Run-time Support for Networked Embedded Systems (MidSens 2011)*. Lisbon, Portugal: ACM, 2011.

[4] ——, "myhealthassistant: A phone-based body sensor network that captures the wearer's exercises throughout the day," in *The 6th International Conference on Body Area Networks*. Beijing, China: ACM Press, November 2011.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, 2003.

[7] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2010.

[8] PowerTutor, "A Power Monitor for Android-Based Mobile Platforms," http://ziyang.eecs.umich.edu/projects/powertutor/, 2012, retrieved on 27 November 2012.

[9] M. Chen, S. Gonzalez, A. Vasilakos, H. Cao, and V. C. Leung, "Body area networks: A survey," *Mob. Netw. Appl.*, Apr. 2011.

[10] P. Neves, M. Stachyra, and J. Rodrigues, "Application of Wireless Sensor Networks to Healthcare Promotion," *Journal of Communications Software and Systems (JCOMSS)*, 2008.

[11] P. Brandão and J. Bacon, "Body sensor networks: can we use them?" in *Proceedings of the International Workshop on Middleware for Pervasive Mobile and Embedded Computing (M-PAC '09)*. ACM, 2009.

[12] A. Waluyo, I. Pek, S. Ying, J. Wu, X. Chen, and W.-S. Yeoh, "Litemwban: A lightweight middleware for wireless body area network," in *5th International Summer School and Symposium on Medical Devices and Biosensors*, 2008.

[13] A. B. Waluyo, W.-S. Yeoh, I. Pek, Y. Yong, and X. Chen, "Mobisense: Mobile body sensor network for ambulatory monitoring," *ACM Trans. Embed. Comput. Syst.*, 2010.

[14] X. Chen, A. Waluyo, I. Pek, and W.-S. Yeoh, "Mobile middleware for wireless body area network," in *Engineering in Medicine and Biology Society (EMBC)*. IEEE, 2010.

[15] S. L. Keoh, N. Dulay, E. Lupu, K. Twidle, A. Schaeffer-Filho, M. Sloman, S. Heeps, S. Strowes, and J. Sventek, "Self-managed cell: A middleware for managing body-sensor networks," in *Mobile and Ubiquitous Systems: Networking Services (MobiQuitous)*, 2007.

[16] J. Sventek, N. Badr, N. Dulay, S. Heeps, E. Lupu, and M. Sloman, "Self-managed cells and their federation," in *17th Conference on Advanced Information Systems Engineering*, 2005.

[17] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo, "Middleware to support sensor network applications," *Network*, 2004.

[18] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner, "Mires: a publish/subscribe middleware for sensor networks," *Personal Ubiquitous Comput.*, Dec. 2005.

[19] V. Jones, A. Van Halteren, N. Dokovsky, G. Koprinkov, J. Peuscher, R. Bults, D. Konstantas, W. Ing, and R. Herzog, *MobiHealth: Mobile Services for Health Professionals, in M-Health: Emerging Mobile Health Systems*. Springer-Verlag New York Inc, 2006.

[20] K. Kang, S. Heo, and C. Bae, "Android/osgi-based mobile healthcare platform," in *7th International Conference on Advanced Information Management and Service (ICIPM)*, 2011.