Journal of Applied Computing Research, 5(1):17-31
January-June 2016
Unisinos - doi: 10.4013/jacr.2016.51.02

Enabling Dynamic Crowdsensing through Models@Runtime[1]

Paulo Cesar Ferreira Melo, Ricardo Couto Antunes da Rocha, Fabio M. Costa
Universidade Federal de Goiás, Instituto de Informática
Alameda das Palmeiras, Quadra D, Campus Samambaia, 74690-900, Goiânia, GO, Brasil
pcfm.inf@gmail.com, ricardo@inf.ufg.br, fmc@inf.ufg.br

Abstract. The complexity of applications in the mobile crowdsensing domain is due to factors such as interoperability among heterogeneous devices, recruiting of devices, collection of data from these devices, and adaptation of application operation in dynamic environments. This paper introduces a platform based on models at runtime (M@RT) for the development of the mobile crowdsensing functionality of applications. The platform supports model-based creation and processing of queries that target a distributed and dynamic set of sensor-capable devices. The paper also presents the results of an evaluation that shows the impact of runtime model processing on the performance of applications in mobile crowdsensing scenarios.

Keywords: participatory sensing, models at runtime, model execution engine, mobile computing.

Introduction

Crowdsensing refers to the opportunistic or participatory use of a large array of sensors embedded in current general purpose mobile devices for the purpose of measuring and mapping interesting phenomena by means of the collaborative sharing of sensors (Ganti *et al*., 2011). The development of mobile crowdsensing applications poses a number of challenges, such as interoperability among different mobile devices, recruiting of appropriate devices to serve as data sources, collection of data from those devices, and runtime adaptation of the applications to work properly in dynamic environments.

A number of platforms for crowdsensing have been developed, such as Medusa (Ra *et al*., 2012), Vita (Chan *et al*., 2013) and MobIoT (Hachem *et al*., 2014). They address challenges such as facilitating application development, supporting efficient and scalable dissemination of sensor data, enabling mobility management of crowdsensing applications, and providing incentives for participatory sensing. However, the programming model of existing platforms hinders the development of dynamic and spontaneous applications when it comes to enabling rapid user-directed development and dynamic change of application behavior. This is a particularly important requirement for ephemeral applications in scenarios such as disaster recovery and environmental monitoring for short-lived purposes.

This paper presents CSVM (CrowdSensing Virtual Machine) (Melo, 2014), a platform driven by models@runtime (Blair *et al*., 2009) that enables the creation and execution of mobile crowdsensing queries by means of the specification and interpretation of models described in a domain-specific modeling language called CSML (CrowdSensing Modeling Language). The use of models@runtime allows the description of the dynamic behavior of applications (Wang *et al*., 2008), including those used for crowdsensing, thus enabling runtime adaptation of such behavior.

In general, the use of a model-based approach enables significant shortening of the semantic gap between the problem to be solved and the platform being used, promoting the use of abstractions that are closer to the problem domain and, thus, more accessible to end users. Specifically in our case, the mod-

[1] This article is an extended version of the paper presented by the authors at the 7th SBCUP -*Simpósio Brasileiro de Computação Ubíqua e Pervasiva,* an event of the 35th *Congresso da Sociedade Brasileira de Computação*, Recife (PE, Brazil), July 20-23, 2015.

els@runtime approach promotes the use of a causally connected representation of both the crowdsensing environment and the queries that are submitted for processing. This enables dynamic adaptation of the crowdsensing environment and applications in response to device mobility and reconfiguration, as well as to the need to change ongoing queries.

The remainder of this paper is organized as follows. *An approach for mobile crowdsensing based on M@RT* presents our motivation scenario and provides an overview of our approach in the domain of mobile crowdsensing. *The CSML modeling language* presents the CSML language, while *CSVM platform architecture* describes the architecture of the CSVM platform, which executes CSML models, while *Implementation* outlines its implementation. *Experiments and evaluation* presents a set of practical experiments that demonstrate the capabilities of the platform, together with results of a performance evaluation of the different phases of model execution. Finally, *Related work* discusses related work, and *Concluding remarks* reviews the main contributions and discusses future work.

An approach for mobile crowdsensing based on M@RT

Mobile crowdsensing environments encompass a variety of applications that need to communicate and exchange data derived from sensors embedded in mobile devices. The essence and major challenges in such environments lie in the amount and diversity of devices, in the dynamic nature of typical scenarios, and in the process of selecting the appropriate devices to fulfill a given request for distributed sensing data (Ganti *et al*., 2011). In general, platforms for crowdsensing need to identify and select devices and their sensors based on requirements previously specified by users and/or developers, typically using a domain-specific language. Such platforms also aim to abstract vendor-specific details of how different physical devices and their sensors are accessed. In addition, some platforms, such as Medusa (Ra *et al*., 2012) and Vita (Chan *et al*., 2013), focus on efficient and scalable dissemination of sensor data and on the economic issue (sensor provision vs. data demand). These solutions assume that a crowdsensing application must be previously planned (in the form of sensor queries) in terms of knowledge of sensor availability and considering a static interest in sensor data.

Provision of sensor data (by devices) is dynamic and, in some cases, unpredictable. Thus, requiring, from the application developer, knowledge of the crowdsensing environment may hinder the development of opportunistic and volatile applications. We argue in favor of the development of platforms for crowdsensing applications that aim at dynamic and spontaneous scenarios, such as in the examples that follow:

- Support for disaster recovery through the recruiting of available sensors, for instance, in situations like flooding and firefighting.
- Support for users in an open-air concert trying to use other sensors spread across the place to build a better concert experience, e.g., to find a spot with a better sound quality.
- Exploratory usage of distributed sensor data by spontaneous applications.

Development of crowdsensing application in such scenarios demands (i) fast application prototyping and setup; (ii) user-friendly discovery of distributed sensor collections; (iii) dynamic update of sensor queries and frequent readjustment of crowdsensing applications; and (iv) a user-centered approach.

In order to overcome these challenges, we propose a platform driven by models@runtime for mobile crowdsensing applications called CSVM (CrowdSensing Virtual Machine). The platform materializes our approach to fulfill the following goals:

- Dynamically adapt to changes in the crowdsensing environment, by adjusting the set of selected devices and sensors in a way that ensures quality of the sensor data requested by applications; and
- Allow users to alter queries on-the-fly, especially in the case of long-running queries, in a way that reflects on how the platform behaves in relation to execution of the query.

The rationale behind the design of CSVM is that the use of models@runtime enables effective programming of crowdsensing applications through a modeling language (CSML). The causal connection between the model and the running system allows both the platform to reproduce the behavior described in a high-level model, and the model to describe runtime changes of the environment. As a consequence, the approach proposed in this work does not require application users and devel-

opers to be specialists in the domain of crowdsensing, as it facilitates the specification and adaptation of queries by means of high-level models that abstract the operational details and the heterogeneity of the environment.

The CSML modeling language

CSML is a domain-specific modeling language for the domain of mobile crowdsensing. It allows the creation and runtime manipulation of models that describe queries and their execution, as well as other elements of the domain. It allows users and developers to concentrate on the declarative features of the crowdsensing tasks, abstracting away the details of how the CML executor carries out those tasks. In its current implementation, the syntax of CML is based on XML.

CSML constructs enable the modeling of two major functionalities required by crowdsensing applications, namely device registration, which integrates the device as part of the crowdsensing environment, and query specification, which allows users to create queries that involve sensor data gathered from multiple devices. These two functionalities are specified in the form of two kinds of sub models, also called schemas: control schema (CS) and data schema (DS). Control schemas are models that represent logical crowdsensing configurations and are further subdivided into environment control schemas (ECS) and query control schemas (QCS). The constructs used to specify schemas are defined in the CSML metamodel, which in turn is defined according to OMG's metamodeling architecture, the Meta-Object Facility (MOF) (OMG, 2008).

An ECS describes the crowdsensing environment by means of the components presented in Table 1. It thus serves as a representation of the devices that are available in the environment, including the individual sensors that are made available by each device and the specification of how their provided sensor data are handled and communicated.

A QCS, in turn, is a model at runtime that specifies one or more queries in terms of the desired types and number of sensors, their location, data collection and combination operations (as defined in Table 1), and the type of notification. As an example, a QCS can be used to describe a query to measure air moisture (sensor type) in a given indoor environment (location) by means of data collected from 10 devices spread across the environment, such that the application is notified whenever the average air moisture reaches a threshold.

Data schemas (DS), in turn, are different from the above two kinds of schemas, which are explicitly created by a user or application developer. A DS is automatically created by the platform as a result of processing a previously created QCS. A DS thus represents an empty form, which contains fields such as value (to contain actual data collected from a sensor), data type of the value, unit of measurement, sensor type, precision, and time of data collection. A DS also contains data that describe a request (which is the act of sending a DS form to a device), such as the type of the request (which specifies how requests are sent – unicast, multicast or broadcast), the notification type (which specifies when replies are sent as response to requests, such as when an event occurs or on a periodic basis), and the identifier of the query that generated the DS. Examples of data and control schemas are presented in *Using the CSML language.*

Using the CSML language

In this section we demonstrate the use of the CSML language to create control schemas (both ECS and QCS). For this purpose, we use the XML-based syntax of CSML, called

Table 1. Components of an ECS model.

ECS Component	Description	Example
Device	Device make and model	A given Android smartphone model #
Sensor list	Sensors made available by the device	Location, temperature
Operation	Operations to collect data from multiple sensors of the same type	Average of temperatures
Combination type	Method of combination of data from sensors of different types	Aggregation of temperature and location
Notification type	Method of sensor data notification	Event-based or periodic

X-CSML. We aim to demonstrate the applicability of CSML models in a typical scenario where crowdsensing functionality is required. The scenario can be described as follows.

"The new manager of a nightclub, among his many responsibilities, is in charge of monitoring the ambient temperature inside the premises whenever the house is open for events. His aim is to maintain a comfortable environment for the customers, as well as to detect hazardous situations, such as the start of a fire. To carry out his task, the manager was instructed to use an app for temperature monitoring that was installed in his smartphone. The app uses the crowdsensing services of the CSVM platform for the collection of temperature data from sensors spread across the environment. In addition, frequent customers of the nightclub are invited to install a corresponding app on their smartphones. They agree to provide sensing information from their smartphones' sensors, whenever they are in the nightclub, in exchange for a safer environment and, obviously, a half-price discount in the admission ticket."

As part of the above scenario, all the involved devices (smartphones belonging to the manager and to the customers) need to register in the CSVM platform so that they are able to submit crowdsensing queries and to provide sensing data to the platform. Figure 1 presents the CSML control schema (represented in X-CSML) that is generated when the nightclub manager requests registration of his device. This control schema is built as a fragment of the environment control schema (ECS) and is submitted to the CSVM platform, where it becomes part of the ECS that represents the entire crowdsensing environment. Similar control schemas are generated and submitted to CSVM whenever a customer registers his or her device.

After registration, the manager is able to specify queries that use the crowdsensing services of the platform. In general, such queries are generated to answer high-level questions such as: "What is the average room temperature at a given part of the nightclub?" Such questions are specified in a user-friendly interface provided by the app that was previously installed in the manager's smartphone. The app then uses client-side functionality of the CSVM platform to convert the question into a more specific query such as *"I need information from 10 sensors of type 'temperature' at a location 'bar'."* We assume the availability of an indoor location system that allows the current location of devices to be determined by the platform. The location system, however, is out of scope in this work.

Figure 2 presents the query control schema that was generated in response to the manager's

```
<?xml version="1.0" encoding="UTF-8"?>
<xcsml>
  <controlSchema>
    <envControlSchema>
      <registration registerID="1">
        <device>
          <type>
            <brand>Samsung</brand>
            <model>Galaxy S5</model>
            <os>Android OS</os>
            <category>Smartphone</category>
          </type>
          <owner>John</owner>
          <deviceSensor>
            <sensorType id="1">temperature</sensorType>
            <sensorType id="2">humidity</sensorType>
            <sensorType id="3">audio</sensorType>
            <sensorType id="4">geolocation</sensorType>
          </deviceSensor>
        </device>
      </registration>
    </envControlSchema>
  </controlSchema>
</xcsml>
```

Figure 1. X-CSML representation of a control schema used to register a device in the CSVM platform.

query. The operation kind (average – avg) and the request type (onDemand) are also specified as part of the query. The next step is to submit the query (the QCS) to the server side of CSVM, which then creates an instance of the QCS by selecting and recruiting the actual devices that will provide sensor data to answer the manager's query. A description of the recruited devices is added to the QCS instance to facilitate its processing by the platform, especially considering the management and runtime adaptation of the set of devices selected to answer each crowdsensing query. The mechanics of QCS instance generation and processing is described in *CSVM platform architecture*. The generated QCS instance is partly shown in Figure 3, which, for brevity, omits the description of some of the recruited devices. A complete version of this QCS instance can be found in Melo (2014).

After generating the QCS instance, the platform then generates a data schema (DS) for the query. Figure 4 presents the X-CSML representation of the data schema, which contains a form to be sent to each recruited device, along with a list of all the devices recruited for the query. The request that is actually sent to the devices is thus an empty form, which should be filled by the device with actual sensor data. Thus, after receiving such a form (i.e., a data schema), each device initiates its part in the monitoring of the environment. The result is a sequence of data schema instances that each device sends to the server side of the platform, containing the requested sensor data. Figure 5 shows an example of a DS instance (a filled form) received from one of the recruited devices. After receiving the DS instances from all the recruited devices, the platform composes a reply that is then sent to the requesting client (the app running on the manager's smartphone). The reply is composed by combining all the received sensor data in the way that was specified in the QCS (Figure 2) and takes the form of a DS instance similar to the one shown in Figure 5.

Note that the CSML language is meant for modeling all aspects of a crowdsensing environment. This includes its direct use by the end-user (to contribute in the creation of the ECS and to create QCSs) and by the platform, which automatically generates CSML models (data schemas) for the processing of queries. The next section describes the processing of CSML models by the platform.

CSVM platform architecture

The architecture of mobile crowdsensing platforms generally comprises two kinds of components: one that represents the devices that are responsible for collecting and propa-

```
<?xml version="1.0" encoding="UTF-8"?>
<xcsml>
   <controlSchema>
     <queryControlSchema>
       <subscription subscriptionID="1">
         <aggregation />
         <sensorTypeRequest id="1">
           <type>temperature</type>
           <quantity>10</quantity>
           <location>boate x</location>
           <operation>avg</operation>
           <request>onDemand</request>
         </sensorTypeRequest>
       </subscription>
     </queryControlSchema>
   </controlSchema>
</xcsml>
```

Figure 2. Query control schema (QCS).

```
<?xml version="1.0" encoding="UTF-8"?>
<xcsml>
   <controlSchema>
     <queryControlSchema>
       <subscription subscriptionID="1">
         <aggregation />
         <sensorTypeRequest id="1">
           <type>temperature</type>
           <quantity>10</quantity>
           <location>boate x</location>
           <operation>avg</operation>
           <request>onDemand</request>
           <device  id="1">
             <type>
               <brand>Samsung</brand>
               <model>Galaxy S5</model>
             </type>
             <owner>John</owner>
           </device>
           ...
           <device  id="20">
             <type>
               <brand>LG</brand>
               <model>Nexus 4</model>
             </type>
             <owner>Taylor</owner>
           </device>
         </sensorTypeRequest>
       </subscription>
     </queryControlSchema>
   </controlSchema>
</xcsml>
```

Figure 3. Query control schema instance.

```
<?xml version="1.0" encoding="UTF-8"?>
<xcsml>
 <dataSchema>
  <request requestID = "1">
  <queryControlSchemaID>1</queryControlSchemaID>
     <type>multicast</type>
     <period>onDemand</period>
     <form>
         <sensorInformation>
           <sensorType>Temperature</sensorType>
           <value></value>
           <dataType></dataType>
           <unit></unit>
           <timeStamp></timeStamp>
           <accuracy></accuracy>
         </sensorInformation>
     </form>
     <deviceRequest>
       <deviceID>1</deviceID>
       <deviceID>2</deviceID>
       <deviceID>3</deviceID>
       <deviceID>4</deviceID>
       <deviceID>5</deviceID>
       <deviceID>6</deviceID>
       <deviceID>7</deviceID>
       <deviceID>8</deviceID>
       <deviceID>9</deviceID>
       <deviceID>10</deviceID>
     </deviceRequest>
   </request>
  </dataSchema>
</xcsml>
```

Figure 4. Data schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xcsml>
 <dataSchema>
   <notification notificationtID = «1»>
     <requestID>1</requestID>
     <type>multicast</type>
     <period>onDemand</period>
     <form>
         <sensorInformation>
           <sensorType>Temperature</sensorType>
           <value>23</value>
           <dataType>Ambient temperature</
dataType>
           <unit>degree Celsius</unit>
           <timeStamp>2014-01-23 23:15:00</
timeStamp>
           <accuracy>3</accuracy>
         </sensorInformation>
     </form>
   </notification>
 </dataSchema>
</xcsml>
```

Figure 5. Data schema instance.

gating sensor data, and another that represents the service in charge of analyzing and processing the sensor data gathered from the devices (Ganti *et al*., 2011). Following this generic approach, this work proposes an architecture comprised by a central component (CSVMProvider), which has a single instance in the system and represents the provider of the crowdsensing service, and a distributed component (CSVM4Dev), which is instantiated in each mobile device that is part of the environment. In the above discussion (*Using the CSML language*), these two components were referred to, respectively, as the server and client sides of the platform. In the remainder of this section, we describe the architecture of these two components, together with a description of how they cooperate to process control and data schemas. We also provide some details about their implementation.

CSVMProvider

The CSVMProvider component is responsible for the interpretation of user models (schemas) in order to determine the actions that need to be taken by the platform so that the user's intention, expressed in the models, is effectively carried out. Those actions are expressed in terms of query functionality and dynamic management of the devices that participate in the crowdsensing environment. The component has a three-layer architecture, as shown in Figure 6 and described next.

Crowdsensing Synthesis Layer (CSS). This layer is responsible for the interpretation of X-CSML models, synthesizing them in the form of commands to be executed by the underlying layers. The declarative definitions in the models (e.g., in query control schemas) are thus transformed into procedural definitions that drive the platform behavior to fulfill the user's intention. This approach, and in fact the entire layered architecture of CSVM, was inspired by the architecture of the CVM platform, as proposed by Clarke *et al*. (2006).

The CSS layer also plays a role in maintaining the model@runtime, which is divided in two parts: one that reflects the structure of currently running queries and another that reflects the state of the environment. The former corresponds to the user-defined QCSs, while the latter refers to the ECS. We first consider the maintenance of the part of the model@runtime that represents queries.

During execution of a query the user can make modifications in its model (QCS), resubmitting it to the platform as an intention to change the functionality of the query. The CSS layer isolates the specific changes by analyzing the differences between the old QCS model and the new one, which results in the generation of commands (for the underlying

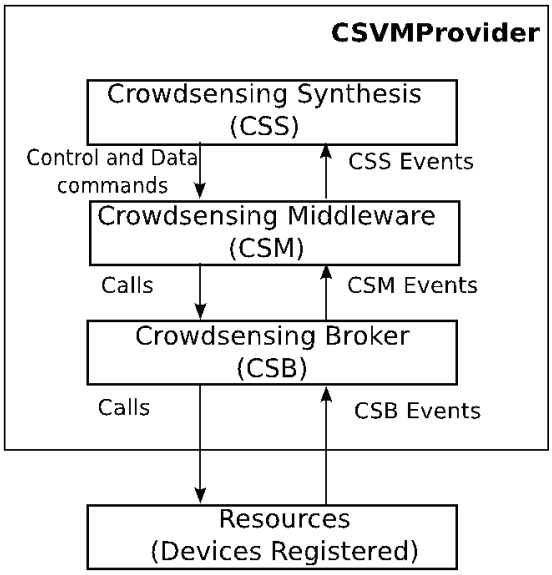

Figure 6. Architecture of the CSVMProvider component.

layer) that correspond to those differences. This is particularly effective for long-running queries (such as one that asks for continuous updates of the sensing information provided by sensors at a given location), for which new requirements may arise during execution. For instance, the user may decide to include a new type of sensor in a query. This change will result in commands for the layer below to initiate the selection and recruitment of further devices (besides those that are already taking part in the query) and/or the enlisting of other sensors belonging to devices that have already been recruited, in order to provide the extra sensing information. In addition, this part of the model@runtime may also be updated as a result of events from the environment. For instance, when all devices that provide a given sensor type leave the environment, the QCSs that referred to that type are changed to reflect the fact, so that users know that the corresponding feature is no longer available.

We now consider the maintenance of the part of the model@runtime that represents the environment. This is usually the result of devices entering and leaving the environment. When a device enters the environment, its registration control schema is processed and incorporated into the ECS. Similarly, when a device leaves, its description is removed from the ECS. Thus, the ECS always reflects the current state of the environment in terms of the devices that are currently available for crowdsensing.

Crowdsensing Middleware layer (CSM). This layer of CSVMProvider is responsible for the selection of devices based on information from the model@runtime, notably information about the capabilities and location of devices. Such information is obtained from the part of the model@runtime that corresponds to the environment: the environment control schema (ECS). The CSM layer interacts with the other two layers by processing commands issued by the top layer (CSS), and by generating lower-level commands to be executed by the layer below. The CSM layer processes CSS commands by matching them with the existing capabilities that are present in the environment (and represented in the ECS). One such command may be "select 10 devices that provide temperature sensors". The processing of all commands generated from a QCS thus results in the generation of a QCS instance (such as the one shown in Figure 3). This QCS instance becomes part of the model@runtime of the query (complementing the part of the model@runtime of the query maintained by the CSS layer). The CSS layer then processes the QCS instance to generate commands for the bottom layer (CSB) to actually recruit each of the selected devices.

The CSM layer also plays a role in the maintenance of the model@runtime, notably with respect to currently running queries. This is necessary due to the volatility of the environment, where devices come and go all the time, which in turn requires continuous update of the QCS instance part of the model@runtime, followed by the generation of commands (for the bottom layer) that effectively adapt the query during its execution. For instance, when a device leaves the environment, all queries to which it contributed with sensing data need to be adapted by selecting substitute devices.

In future versions of the platform, the CSM layer will also be responsible for the addition of non-functional properties in query processing and device registration, especially in the case of security aspects, such as confidentiality, integrity and authentication.

Crowdsensing Broker layer (CSB). This layer is in charge of the actual interaction with the devices that provide sensing capabilities for the platform. Its main responsibility is to recruit the devices selected by the CSM layer (in response to commands received from that layer). This involves the following three functionalities: masking the heterogeneity of the devices by means of adaptors that allow seam-

less access to different kinds of devices and their resources (e.g., Android and iOS smartphones); monitoring of devices to detect when they leave the environment (in such cases, the CSB layer generates events to the upper layer so that it may select a substitute device); and communication with the mobile devices.

CSVM4Dev

This is the component of CSVM that runs on the devices and allows them to be integrated in the platform. It also provides the user interface that allows the modeling of queries (i.e., the creation of query control schemas). Similarly to CSVMProvider, it has a layered architecture, as shown in Figure 7, which facilitates its implementation for different mobile device technologies. Differently from CSVMProvider, CSVM4Dev has an additional layer, the **CrowdSensing Application layer (CSApp)**, which provides a user interface (GUI) and a programming interface (API) for end-users and applications to interact with the platform. In particular, this layer provides a modeling environment for users to create and submit device registration models and query models (QCS). The CSApp user interface also exposes events that could not be handled by the lower layers, such as when a given user query cannot be satisfied by the platform, e.g., when there are not enough devices to provide a given type of sensor. As a response, the user can either change (and resubmit) the QCS for the query or completely withdraw the query.

The other layers of CSVM4Dev have similar, but more limited, responsibilities compared to their CSVMProvider counterparts. The CSS layer is in charge of validating the models created by users and converting them into text-based X-CSML models before sending them to the CSVMProvider component. The CSM layer, in turn, is in charge of applying non-functional properties to queries and device registration, such as security and privacy. Its current implementation, though, deals only with privacy issues, by filtering any attempt to access sensors that were not explicitly exposed by the user when he or she registered the device. Last but not least, the CSB layer is responsible for the access to the resources of the device (notably its embedded sensors) and for communication with the CSVMProvider component.

The remainder of this section describes the interaction protocols that these components follow in order to perform the platform's crowdsensing functions.

Device registration

Device registration precedes all other functionalities of the platform and is illustrated in Figure 8 in terms of the interactions between the CSVM4Dev and CSVMProvider components.

The first stage comprises registration of the device with the communication service so that it is enabled to receive queries sent by CSVMProvider (steps *r1* and *r2* in Figure 8). Next, the user must prepare the device for registration by specifying a model (a control schema) that expresses the device's characteristics (make and model, owner ID, and the sensors that will be available for crowdsensing). CSVM4Dev then sends the device's control schema to CSVMProvider as part of a registration request (step *r3*). CSVMProvider then validates the control schema (to verify that the information it contains is complete) and registers the device in its repository (step *r4*), effectively

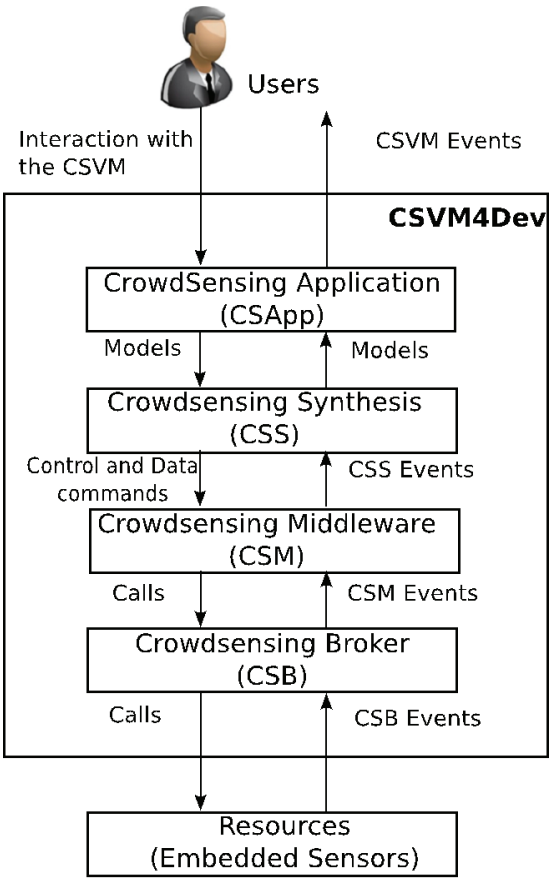

Figure 7. Layered architecture of the CSVM4Dev component.

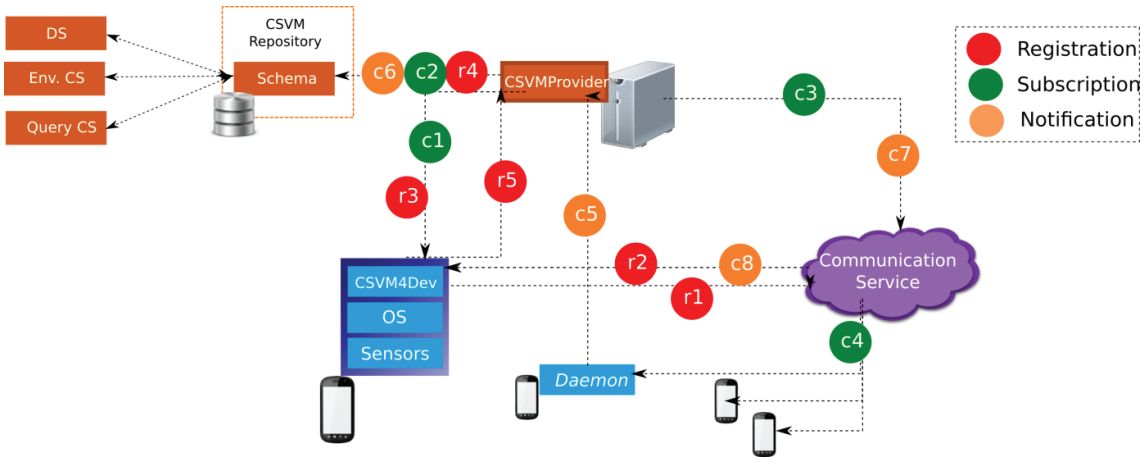

Figure 8. Device registration and query processing.

adding the device's description to the environment control schema (ECS). Finally, the successful registration is notified to the device (step *r5*).

Once registered, the device is enabled to submit crowdsensing queries to the platform, as well as to collaboratively participate in the provision of sensor data to answer crowdsensing queries made by users of other registered devices.

Query processing

The main service in the domain of crowdsensing corresponds to the specification and processing of queries. A query refers to a set of sensors, specified in terms of their attributes (type of sensor, location and quantity). Query processing is also illustrated in Figure 8, which represents the phases of query submission and notification of the requested data, which will be detailed next.

Query submission (step *c1*) refers to the initial steps in the processing of a query (QCS) specified by the user. It exposes the user's interest in sensor data in a particular location, noting that a single query can refer to more than one type of sensor and can combine different types of data to derive compound sensing information.

After receiving a query, CSVMProvider interprets it, generating a series of transactions to get device information from the ECS stored in the repository (step *c2*). This information is used to select a set of devices to satisfy the query, effectively resulting in the generation of a QCS instance, which is the model@runtime that will direct the remainder of the query execution. The next step involves the generation of a data schema (DS), which is sent in the form of a request to the CSVM4Dev component running on each of the selected devices (steps *c3* and *c4*). As a response to these requests, each CSVM4Dev obtains the requested sensor data and sends a notification to CSVMProvider. This phase is described next.

Notification refers to the final phase of query processing. CSVM4Dev processes the received DS and accesses the sensors specified in it to capture the requested sensor data. It then encapsulates the sensor data in a DS instance, which is then notified back to CSVMProvider (step *c5*). As it receives the notifications from the selected devices, CSVMProvider applies the combination operation associated to each of the involved sensor types, such as the calculation of an average. At completion of this processing, CSVMProvider builds a DS instance containing the result of the query and sends it to the device that submitted the query (steps *c7* and *c8*). Finally, the query result is presented by CSVM4Dev to the user or application that generated the query.

Implementation

In this section we briefly describe some aspects of a proof-of-concept prototype of the CSVM platform. The prototype comprises an implementation of the CSVMProvider and CSVM4Dev components. CSVM4Dev was implemented for the Android mobile operating system. Communication between CSVMProvider and CSVM4Dev was implemented using two different technologies: RESTfull for communication from CSVM4Dev to CSVMProvider,

and GCM (Google Cloud Messaging) for communication from CSVMProvider to CSVM-4Dev. The latter was chosen because it is the de facto standard for communication with Android mobile devices, and also because it supports device mobility in a seamless way.

CSVMProvider was implemented in Java. Its implementation is encapsulated as a Web service that provides a RESTful interface (using the JAX-RS API and JSON for the formatting of communicated data). CSVMProvider is executed on top of an Apache Tomcat container.

Persistence of the data and control schemas was achieved using the SQLite library, which was used to implement the repository referred to in *CSVM platform architecture*. The mapping between Java objects and the relational MySQL database is performed using Hibernate.

The complete implementation comprises 85 Java classes that implement the services and processes described in *CSVM platform architecture*.

Experiments and evaluation

In this section we present a series of experiments that provide the basis for a performance evaluation of the platform. The experiment uses synthetic (as opposed to real) scenarios in order to highlight important aspects of crowdsensing performance, enabling a performance evaluation of the critical phases of query processing at different scales. The aim is to show that the overhead imposed by model generation and interpretation can be reasonably tolerated, even in extreme scaling conditions, given the performance requirements of typical mobile crowdsensing applications.

The two phases of crowdsensing query processing considered in this evaluation are the interpretation of query control schema (QCS) and the processing and aggregation of data schemas (DS). The experiment simulates an environment with 100 mobile devices. Each simulated device has 15 different sensor types and is located at a specified logical location. The environment control schema (ECS) containing the description of all the devices was pre-loaded in the repository. Four variants of the experiment were executed:

Experiment 1: Processing of the QCS varying the number of recruited devices for a single sensor type.
Experiment 2: Processing of the QCS varying the number of sensor types.
Experiment 3: Processing and aggregation of a DS varying the number of devices for a single sensor type.
Experiment 4: Processing and aggregation of a DS varying the number of sensor types.

The experiment was set up on a computer with an AMD quad-core 2.4GHz processor, 4GB RAM and Windows 8.1 64 bits to run the CSVMProvider component, and a Samsung Galaxy SIII smartphone with Android 4.3 as the mobile device to run the CSVM4Dev component. All variations of the experiment are

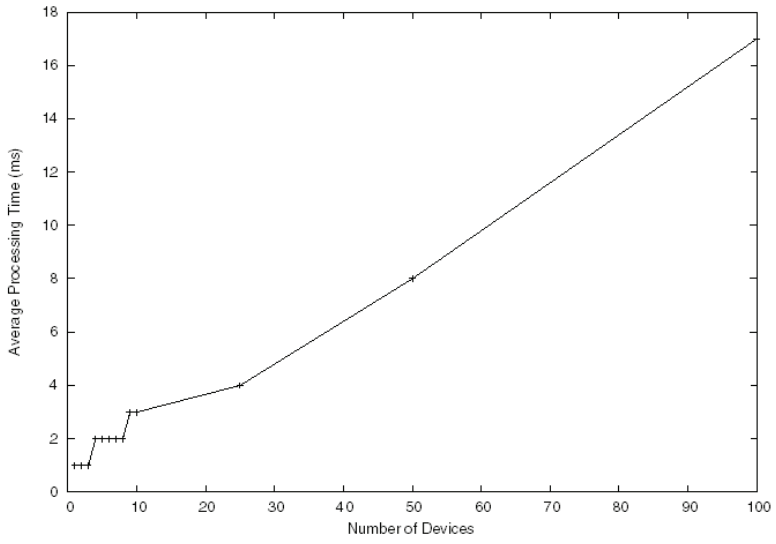

Figure 9. Query processing time as a function of the number of devices involved in the query.

thus carried out using a single mobile device, which is used to simulate the other devices when necessary. This arrangement does not compromise the results, as the experiments only evaluate the model processing steps that occur in the CSVMProvider component, without considering the steps that are carried out in the mobile devices and the communication with them.

Each experiment was carried out for 10 different configurations, varying either the number of devices or the number of sensor types involved in a query, as described above. For each such configuration, 10 runs were carried out, yielding a total of 100 runs for each experiment. The average execution time for each configuration was computed after eliminating the outlier results, which contributed to minimize the error in the experiments. In addition, we also subtracted the time taken for database access from the total execution time of each run of the experiment. The rationale behind this is the fact that database access proved to be a major bottleneck, which points to the need for a lighter approach to implement the mod-

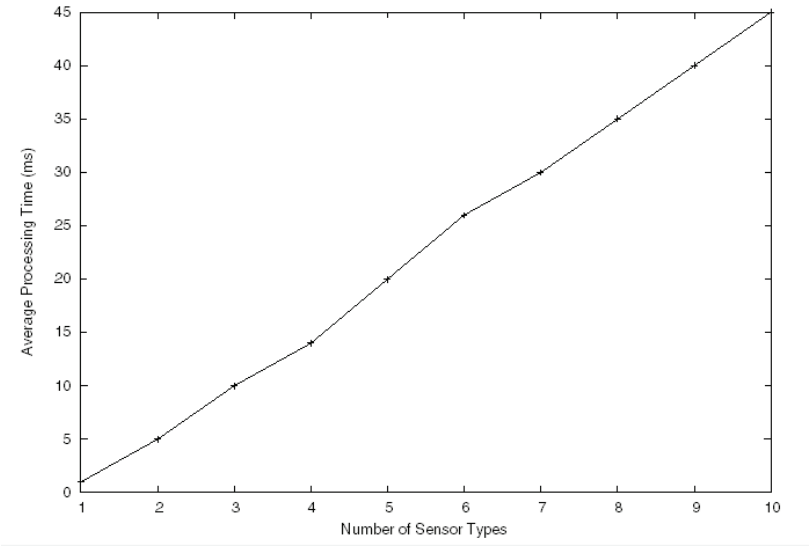

Figure 10. Query processing time as a function of the number of sensor types specified in the query.

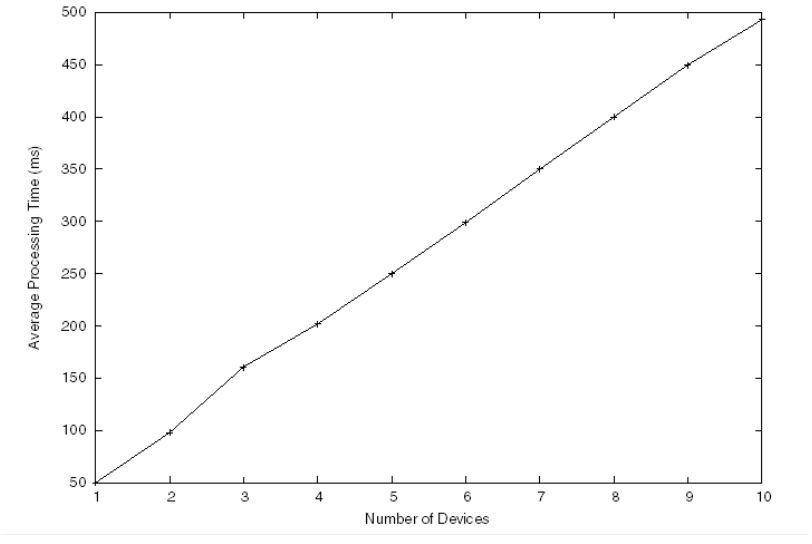

Figure 11. Time for processing and aggregation of a data schema as a function of the number of devices.

el@runtime repository in our future work. Thus, we opted for isolating the time that was strictly consumed by model processing tasks. The results are presented in Figure 9 through Figure 12.

Figures 9 and 10 correspond to experiments 1 and 2 and thus present the average time for processing a query, respectively, as a function of the number of devices (considering a single sensor type) and as a function of the number of sensor types specified in the query. In both experiments, the processing time consistently increases as the other two variables (number of devices and number of sensor types) increase. In the first example (Figure 9) we increase the number of devices more rapidly at the end (25, 50 and 100), only to show that the increase in processing time remains linear, which provides a good indication that query processing scales to large numbers of devices. The second experiment (Figure 10) also produced a linear processing time for queries when the number of sensor types increases.

The graphs in Figures 11 and 12, in turn, present the average time to process sensor data received from the devices (in the form of data schema instance notifications), again as a function of the number of devices and the number of sensor types, respectively. As in the previous experiments, processing time consistently increases as the other two variables increase. As a matter of comparison, however, processing times are an order of magnitude higher than in the previous experiments. This is due to the fact that the amount of operations performed by CSVMProvider in this case (for the combination of sensor data from multiple data schema instances) is much higher than for the interpretation of query control schemas. Nevertheless, the increase is linear in both cases.

The results allow us to draw conclusions about the degree of scalability of CSVM in its current implementation. Although the experiments show consistent increase in the processing time for both queries and reply notifications, this increase is linear, which provides reasonable scalability. Furthermore, within reasonable scaling limits, processing time remains within acceptable bounds for applications that can tolerate response times in the order of a few seconds, which is a typical case for the class of mobile crowdsensing applications that we target (and which are represented by the scenario described in *Using the CSML language*), as well as for a class of medium-scale crowdsensing applications commonly found in the literature (Mathur *et al*., 2010). Nevertheless, for future work we plan to investigate techniques to improve the performance, especially for the processing of reply notifications. We note that the current implementation was developed as a proof-of-concept prototype for model-driven control of crowdsensing functionality, with no specific considerations for performance, which means that there is room for performance improvements.

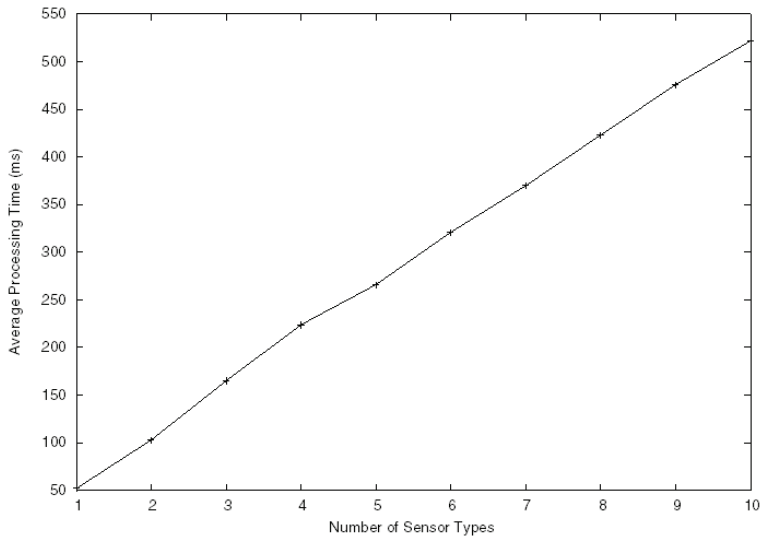

Figure 12. Time for processing and aggregating a data schema as a function of the number of sensor types.

Related work

In this section we compare the CSVM approach with related work in platforms for crowdsensing (*Platforms for Crowdsensing*) and model execution engines (*Model-based Virtual Machines*).

Platforms for crowdsensing

In our comparison we only considered platforms that provide a complete infrastructure for mobile crowdsensing, including an approach for application development, data provision on each sensor's host and a distributed architecture for disseminating data to interested applications.

We considered three representatives of the state-of-the-art platforms for crowdsensing: Medusa (Ra *et al*., 2012), Vita (Chan *et al*., 2013) and MobIoT (Hachem *et al*., 2014). They focus on different aspects of crowdsensing such as incentives for data publishing (Medusa), task management (Vita), and optimal query execution (MobIoT).

Medusa (Ra *et al*., 2012) adopts a domain-specific language based on XML, called MedScript, for the specification of crowdsensing queries. It allows the description of queries in terms of a series of stages (actions), which are connected by flow control elements. In fact, a crowdsensing application describes a control-flow that connects sensor providers and users through a network of sensor aggregators. Medusa offers an infrastructure to deal with the economic issue between providers and consumers, enabling users to recruit sensor providers and to offer incentives for participatory sensing.

Vita (Chan *et al*., 2013) is a platform for cyber-physical systems that provides a graphical user interface for the specification of crowdsensing queries. The Vita platform offers a means for the efficient management of tasks and resources for collecting and aggregating sensor data, automatically dealing with task failures. The selection of sensor data is based on the computational resources available in the devices.

The MobIoT middleware (Hachem *et al*., 2014) allows the specification of crowdsensing queries by means of a language based on SQL and TinyDB. Its aim is to improve scalability by selecting the minimum number of devices to provide the necessary sensing resolution within a particular coverage area. MobIoT performs device selection based on the mobility patterns of devices that satisfy the required monitoring coverage. MobIoT crowdsensing applications are statically developed, meaning that changes in crowdsensing queries require application redevelopment.

In order to enable dynamic crowdsensing, we argue that platforms should provide support for strong decoupling between consumers and providers of sensing data, avoiding static dependency between consumer and provider or sensing data types. Table 2 shows a comparison of the above-described platforms with CSVM.

In Vita, sensor data is accessed through a RESTful interface, provided by previously deployed applications. In Medusa, consumption of sensor data depends on the previous recruitment of devices and on a previous agreement to provide data. In both models, users must have previous knowledge of sensing data types. Existing applications must be redeveloped, according to each platform's application model, in order to incorporate dynamic changes in the crowdsensing environment. Both platforms offer repositories for sensor metadata, although the development model remains static.

In MobIoT, crowdsensing queries are structurally similar to SQL, thus requiring knowledge of the crowdsensing data types prior to application development. For this purpose, MobIoT offers a knowledge repository where users can browse ontologies of crowdsensing data.

Queries for sensor consumption in Vita are dependent on previously defined publishing applications. Thus, compared with CSVM, Vita proposes a more rigid approach to the development of applications, which is coherent with the proposal of task management, to the detriment of a more dynamic behavior. The creation of new queries in Medusa requires new recruitments (as in new applications) and depends on the acceptance of the task by the sensing device's owner. As a consequence, development of crowdsensing applications in those platforms may be slower than in CSVM.

On the other hand, the Vita and Medusa platforms provide lightweight services for mobile devices, including services to deal with resource consumption and privacy concerns. CSVM deals with sensing data in an ad hoc means, which may hinder the usage of complex data. Vita, Medusa and MobIoT, in turn, are more suitable for well-structured (as opposed to ad hoc) crowdsensing environments.

Table 2. Comparison of CSVM with other platforms for crowdsensing.

	Crowdsensing Platform			
Aspect	**Medusa**	**Vita**	**MobIoT**	**CSVM**
Concerns/ goals	collaboration model; dissemination and economic efficiency	resource and task efficiency	optimized sensor queries (number and physical area of involved devices)	spontaneous and dynamic applications
Approach	user-centric	application-centric	middleware-centric	user-centric
Crowdsensing application	control-flow between providers-consumers	traditional applications; framework-based	traditional applications; middleware-based	dynamic runnable model
Development model	dynamic through recruitment	static	static	dynamic
Application language	MedScript	Java-based (Android)	Java	CSML
Interaction model	stages-connectors	SOA/REST	SOA/REST passive devices	VM-server-VM
Collaboration model	consumers recruit providers for tasks	application-oriented	middleware-oriented	consumers incorporate published sensing data
Decoupling consumer from provider	partial	yes	yes	yes
Architecture	cloud-based	framework and cloud-based	central/cloud service	VM based
Distribution	highly distributed	cloud dependent	requests (query execution)	provider

Model-based Virtual Machines

With respect to the model interpretation techniques used in CSVM, two model execution engines are noteworthy. CVM (Communication Virtual Machine) (Deng *et al*., 2008) is a platform for model-driven specification and execution of user-centric communication services. MGridVM (MicroGrid Virtual Machine) (Allison *et al*., 2011) is a model execution engine based on the principles of CVM but targeting the domain of smart power microgrids. CVM performs the processing of models specified in a DSML called Communication Modeling Language (CML) (Clarke *et al*., 2006), while MGridVM proposes a DSML for the domain of microgrids called MGridML. In a similar way, our work proposes a DSML for the domain of mobile crowdsensing (CSML), together with its model execution engine (CSVM).

In CVM, models@runtime are employed to adapt an ongoing communication session to satisfy new user requirements. Similarly, in MGridVM, models@runtime enable specific properties to be defined by the user and employed in a dynamic way to control and manage the elements of a microgrid. CSVM follows a similar approach, in which user-defined models are employed to describe and adapt crowdsensing functionality. From an architecture point of view, while in CVM all nodes execute an identical instance of the platform, MGridVM has a centralized architecture, in which the execution engine runs in the microgrid controller. CSVM takes a hybrid approach, with a centralized service provider (CSVMProvider) and a distributed, lighter, configuration (CSVM4Dev) that runs on each participating mobile device.

Concluding remarks

In this paper, we present an architecture for mobile crowdsensing based on a DSML, called CSML, and its accompanying virtual machine, CSVM, which together allow the specification, execution and management of models@runtime that describe a crowdsensing environment and related functionalities. The CSML language enables the modeling of the devices

that participate in the environment (by allowing access to their sensing capabilities), as well as the modeling of crowdsensing queries by the user. CSVM in turn is a model execution engine that allows execution and dynamic adaptation of crowdsensing functionality in terms of models@runtime. CSVM has a layered architecture that decouples the different stages of model processing, allowing the use of high-level user-defined models to drive the sensing capabilities of a heterogeneous and dynamic set of mobile devices spread across the crowdsensing environment. The functionalities provided by the platform can be used directly by end-users or as part of applications that need distributed sensing information from voluntary devices.

The main contribution of this work was the demonstration of the feasibility of a models@runtime approach for the mobile crowdsensing domain. The platform uses models@runtime to keep an up-to-date representation of a crowdsensing environment and related queries. It enables the use of a high-level modeling language to create complex queries involving a number of sensors of different kinds embedded in the mobile devices of users spread across a particular physical environment. The platform also allows dynamic adaptation in the case of long-running queries as a result of changes in both the environment and user requirements. Future work includes the enhancement of the security of crowdsensing applications, especially regarding authentication and privacy concerns, as well as performance and scalability improvements.

Acknowledgements

We thank FAPEG (Fundação de Amparo à Pesquisa do Estado de Goiás) and CNPq (Grant 473939/2012-6) for partly funding the work presented in this paper.

References

ALLISON, M.; ALLEN, A.A.; YANG, Z.; CLARKE, P.J. 2011. A software engineering approach to user-driven control of the microgrid. *In:* Software Engineering and Knowledge Engineering (SEKE 2011), Miami Beach, 2011. *Proceedings…* Knowledge Systems Institute, p. 59-64.

BLAIR, G.; BENCOMO, N.; FRANCE, R.B. 2009. Models@run.time. *Computer*, **42**(10):22-27. http://dx.doi.org/10.1109/MC.2009.326

CHAN, H.; CHU, T.; LEUNG, V. 2013. Vita: A crowdsensing-oriented mobile cyber physical system. *IEEE Transactions on Emerging Topics in Computing*, **1**(1):148-165.

http://dx.doi.org/10.1109/TETC.2013.2273359

CLARKE, P.J.; HRISTIDIS, V.; WANG, Y.; PRABAKAR, N.; DENG, Y. 2006. A declarative approach for specifying user-centric communication. *In:* International Symposium on Collaborative Technologies and Systems (CTS 2006), Las Vegas, 2006. *Proceedings…* IEEE Computer Society Press, Los Alamitos, p. 89-98.

http://dx.doi.org/10.1109/CTS.2006.6

DENG, Y.; SADJADI, M.S.; CLARKE, P. J., HRISTIDIS, V.; RANGASWAMI, R.; WANG, Y. 2008. CVM – A communication virtual machine. *Journal of Systems and Software*, **81**(10):1640-1662. http://dx.doi.org/10.1016/j.jss.2008.02.020

GANTI, R.K.; YE, F.; LEI, H. 2011. Mobile crowdsensing: Current state and future challenges. *Communications Magazine*, **49**(11):32-39.

http://dx.doi.org/10.1109/MCOM.2011.6069707

HACHEM, S.; PATHAK, A.; ISSARNY, V. 2014. Service-oriented middleware for large-scale mobile participatory sensing. *Pervasive and Mobile Computing*, **10**:66-82.

http://dx.doi.org/10.1016/j.pmcj.2013.10.010

MATHUR, S.; JIN, T.; KASTURIRANGAN, N.; CHANDRASEKARAN, J.; XUE, W.; GRUTESER, M.; TRAPPE, W. 2010. Parknet: Drive-by sensing of road-side parking statistics. *In:* International Conference on Mobile Systems, Applications, and Services (MobiSys '10), 8, New York, 2010. *Proceedings…* ACM, p. 123-136. http://dx.doi.org/10.1145/1814433.1814448

MELO, P.C.F. 2014. *CSVM: Uma plataforma para crowdsensing móvel dirigida por modelos em tempo de execução.* Goiânia, GO. Master's thesis. Universidade Federal de Goiás, 147 p.

OMG. 2008. Meta Object Facility (MOF) 2.0 query/view/transformation specification. Final Adopted Specification (November 2005).

RA, M.-R.; LIU, B.; LA PORTA, T.F.; GOVINDAN, R. 2012. Medusa: A programming framework for crowd-sensing applications. *In:* International Conference on Mobile Systems, Applications, and Services, 10, New York, 2012. *Proceedings…* ACM, p. 337-350.

http://dx.doi.org/10.1145/2307636.2307668

WANG, Y.; CLARKE, P.J.; WU, Y.; ALLEN, A.; DENG, Y. 2008. Runtime models to support user-centric communication. *In:* International Workshop on Models@runtime, 3, Toulouse, 2008. *Proceedings…* Technical Report COMP-005-2008, Computing Department, Lancaster University, Lancaster, p. 77-86.

Submitted on September 14, 2015

Accepted on January 6, 2016