

Building a platform to collect crowdsensing data. Preliminary considerations

S. George, M. Ganzha, M. Paprzycki, S. Fidanova, and I. Lirkov

Citation: [AIP Conference Proceedings](#) **1895**, 100002 (2017);

View online: <https://doi.org/10.1063/1.5007406>

View Table of Contents: <http://aip.scitation.org/toc/apc/1895/1>

Published by the [American Institute of Physics](#)

Building a Platform to Collect Crowdsensing Data. Preliminary Considerations

S. George^{1,a)}, M. Ganzha^{1,2,b)}, M. Paprzycki^{2,3,c)}, S. Fidanova^{4,d)} and I. Lirkov^{4,e)}

¹*Faculty of Mathematics and Information Science, Warsaw University of Technology, Poland*

²*Systems Research Institute, Polish Academy of Sciences, Warsaw, Poland*

³*Warsaw Management Academy, Poland*

⁴*Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, Sofia, Bulgaria*

a)sando.csc@gmail.com

b)m.ganzha@mini.pw.edu.pl

c)paprzyck@ibspan.waw.pl

d)stefka@parallel.bas.bg

e)ivan@parallel.bas.bg

Abstract. Recent years have seen growing interest in collecting and processing sensor data, in distributed mobile environments. In this context, two, somewhat contradictory, trends have emerged: (1) growing popularity of crowdsourcing-type mechanisms, for (sensor) data collection, and (2) collecting sensed data in data “silos”, which are not only unavailable to “outsiders”, but most often incompatible, thus reducing their usability for data mining. Given these limitations in data accessibility, and compatibility, enormous potential for knowledge discovery is lost. To counter this trend, we propose a generic, adaptive, system that will allow voluntary participation in arbitrary crowdsensing initiatives, with the output stored in a standard data format. The system utilizes a rule-based multiagent approach to instructing sensors when to make readings and how to, if necessary, preprocess them, before sharing the data with user-selected initiatives. The initial version of the system has been implemented, and tested in artificial use case scenario.

INTRODUCTION

Leveraging the ability to collect data by pervasive, sensor equipped, mobile devices (often referred to as *Crowdsensing*) has attracted significant attention from mobile computing researchers, seeing applications in, e.g., environmental [1, 2, 3, 4], infrastructure [5, 6, 7] and social [8, 9] scenarios.

Crowdsensing is generally thought to exist in two “styles”. These are: *participatory* crowdsensing and *opportunistic* crowdsensing [10]. In *participatory crowdsensing*, users have awareness of, and are involved in, the sensing process. As an example of a purely participatory crowdsensing scenario, consider the case of **Alice**, who is a graduate student. She is interested in implementing a crowdsensing campaign to gain insight into how city dwellers view various artistic installations around her city. Contributors are prompted for their opinions, whenever objects of interest are found in their vicinity. No data, other than such feedback, is to be sent / collected.

Opportunistic crowdsensing, on the other hand, is a more autonomous process, with minimal user involvement. Here, let us assume that **Bob** is the owner of a logistics company. He would like to use crowdsensing to perform an analysis of routes taken by company vehicles during daily operations. Except for being in possession of company issued smartphones, drivers are not involved in the sensing process. This is an example of a purely opportunistic crowdsensing scenario.

Crowdsensing campaigns may also use a *hybrid approach*. Consider the case of **Eve**, who is an urban planner. She works in a project to upgrade her city’s public transportation infrastructure. To gather supporting data, and to gain insight into the current commute patterns in her city, she would like to use crowdsensing to check commuter density, across the city, during rush hours. In addition, she plans to prompt contributors to provide feedback concerning their

commute experiences. This is an example of a hybrid approach to crowdsensing, as collected is both the data sent by the smart phones and the data volunteered by participants.

Considering that crowdsensing usually spans a spectrum from participatory to opportunistic, [10] coined the term *mobile crowdsensing* (MCS) to refer to a broad range of paradigms that utilize both styles. Next, [11] further elaborated on MCS and highlighted its roots in participatory sensing [12], while [13] expanded this concept into *mobile crowd sensing and computing* (MCSC), thereby highlighting the increased computing power possessed by the current generation of mobile devices. Since then, the field has seen a steady increase in publications across the major journals. Figure 1 is an illustration of this phenomenon. Here, data was compiled by reviewing contributions returned while searching for “*Mobile Crowdsensing*” or “*Mobile Crowd Sensing*”, in the ACM DL, IEEE Xplore and Springer Link digital libraries. Selected papers were examined for relevance, and those that were deemed relevant were tallied.

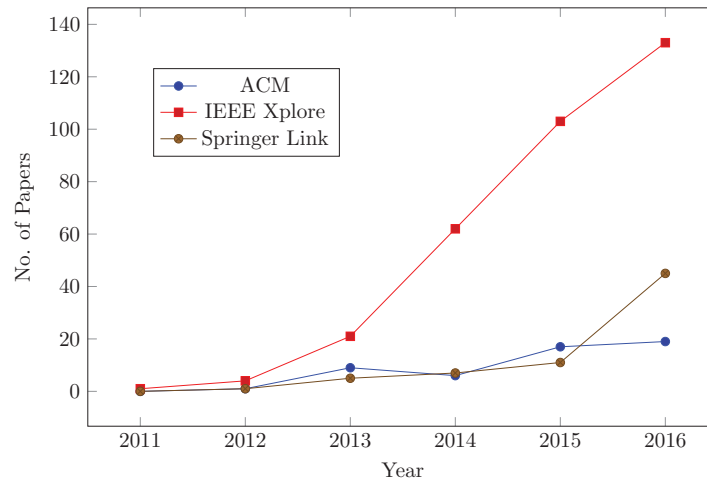


FIGURE 1. Papers mentioning “*Mobile Crowdsensing*” or “*Mobile Crowd Sensing*”

PROBLEM TO BE ADDRESSED

Given the growing prevalence of MCSC systems, in academic and industrial applications, there exists a pronounced need for a generic platform that could be quickly adapted to different scenarios, encountered when implementing such systems. Such a platform would reduce the time and cost associated with prototyping systems in the MCSC domain and would, therefore, speed up the validation of system designs. In this context, our work is focused on three processes, namely, *data gathering and integration*, *local preprocessing of data* and *software defined sensing*, and the concepts of *value-added applications* and sensing coordination via a *task specification language*. Let us now briefly describe each one of them.

1. *Data gathering* involves aggregating data from multiple sources. *Data integration*, on the other hand, deals with combining the gathered data into valuable information. The two tasks are not necessarily performed by the same application. For example, our characters, Alice, Bob and Eve may all have different data management workflows for their intended MCSC campaigns. Alice may wish to make the resulting data available for download and use within Excel worksheets, while Bob and Eve may be more comfortable using applications developed using Matlab-like suite of tools.
2. *Local Preprocessing of data*, as a concept, originates from the increasing computing power of smart devices, and involves data transformations that are performed by the sensing devices, before forwarding the data to an aggregator. For example, if Alice’s application needs to calculate the distance of a data contributor from a given artistic installation (and use it further on the device), she may require that the device completes appropriate calculations, to avoid sending data to a central server and back.
3. *Software defined sensing* involves “soft sensors” that are capable of detecting aspects of the environment that cannot be directly detected by hardware sensors. For example, consider a situation where a study needs to be

conducted, involving a number of applications running on a mobile device throughout the day. Since hardware sensors do not exist for such phenomena, a sensor fully defined in software is a viable alternative. Here, note that in the same way platforms such as Android use a common interface to access all sensors, it would be helpful if all software defined sensors presented a common interface for accessing readings and return data values in a standard format. As a matter of fact, it would be highly desirable if such interface would be “as close as possible”, in its form, to the one exposed by the operating system of the mobile device.

4. *Value-added applications* are applications that give device owners insight into what is being done with the data that is collected. For instance, in Bob’s campaign, it may be beneficial for the drivers to have access to the generated data, superimposed on a real time traffic congestion map, to better assist them in route planning. Bob may develop a plugin application to satisfy this need. However, allowing such applications a “carte blanche access” to the data generated, poses numerous security and privacy risks. A system utilizing a plugin architecture must, therefore, provide a standard, controlled, interface for them to access the data.
5. A *Task Specification Language (TSL)*, in the context of this research, is an approach to communicating “how” sensors are to be read and the resulting data processed.

PROPOSED SOLUTION

Upon an in-depth analysis of possible ways of instantiating the needed platform (details of which are omitted for brevity), we have decided to develop a solution in the form of a *multiagent system (MAS)*. MASs are distributed computing environments consisting of subsystems that are capable of flexible autonomous action in dynamic and unpredictable domains [14]. Here, the most typical conceptualization of the approach is based on the following metaphor: each identified subsystem is referred to as an *agent*. Given the distributed nature of both MCSC and MASs, the fusion of the paradigms was a natural choice in designing an architecture that supports autonomous, and adaptive, behaviour on the part of sensing devices. The proposed architecture is rule-based, and founded on five main concepts: *data collection channels*, *sensor rules*, *orchestration agents*, *local agents* and an *aggregation server*. Figure 2 introduces the top-level schematics of the proposed architecture.

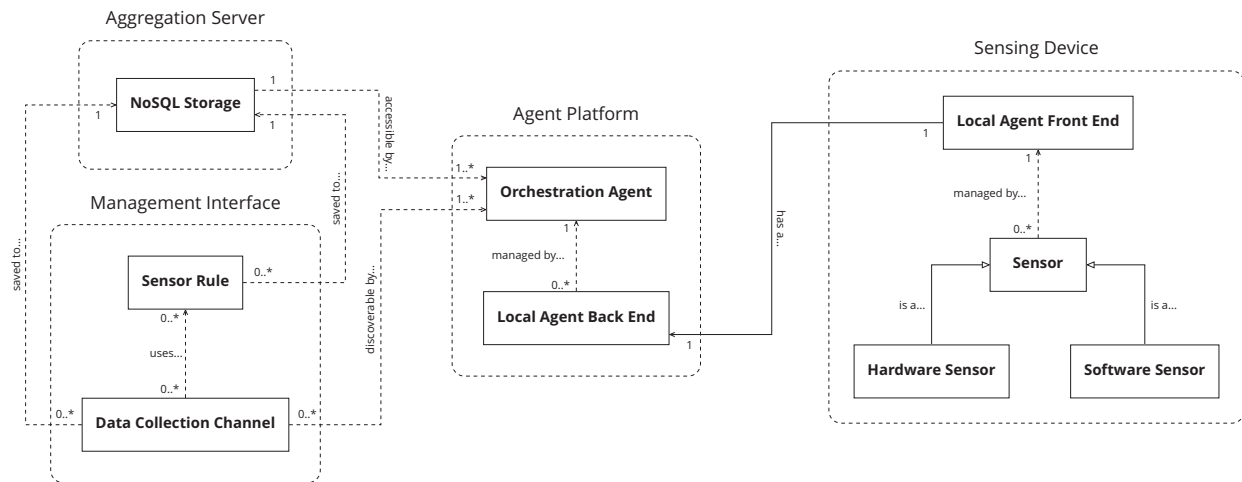


FIGURE 2. Proposed system architecture

Let us now describe each of defined components of the proposed architecture.

1. *Data Collection Channels (DCCs)* – are abstractions of MCSC campaigns. Each campaign, therefore, has an associated DCC and, also, has a set of *Sensor Rules* that define the specifics of the campaign.
2. *Sensor Rules (SRs)* – are “wrappers for knowledge” concerning how a given sensor type is to be handled (what to read, how often, *etc.*) and, if necessary, how the collected data is to be preprocessed. SRs and DCCs share a many-to-many relationship. In other words, the same SR may be used in multiple campaigns, while a single DCC may collect data from multiple sensors (while applying multiple rules).

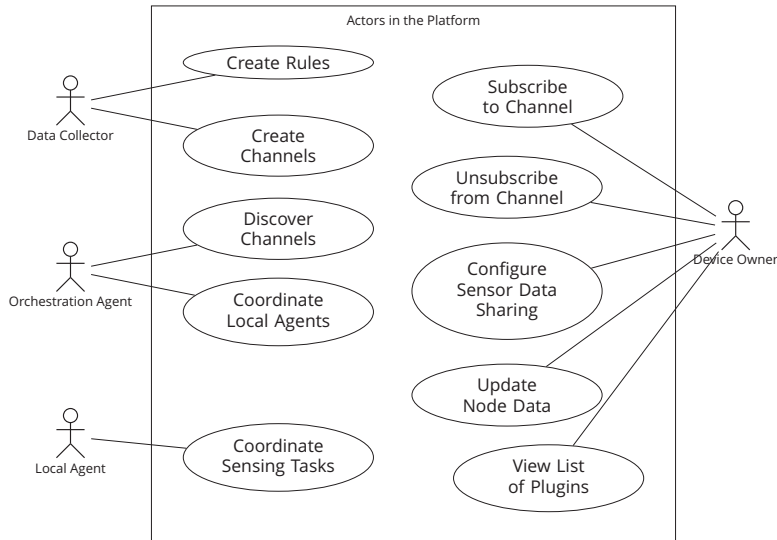


FIGURE 3. Use case diagram for platform actors

3. *Orchestration Agents* (OAs) – live within an agent platform, initialized in the cloud. They are responsible for coordinating communication in the system (data transfers), by being aware of existing DCCs, their associated SRs, and the subscribed devices.
4. The *Aggregation Server* (AS) – is a document storage server, interfaced by the OAs, where all sensing data (deemed worthy of saving) is accumulated. When sensor data is to be passed back to the AS, it is first passed to an appropriate OA, which, in turn, stores the data into the AS.
5. *Local Agents* (LAs) – are entities that are split between the agent platform and the sensing devices and act as conduits for communication between devices and the wider sensor network. Additionally, they are aware of the DCCs that a device belongs to, and, inherently, SRs governing the behaviour required by these DCCs.

The parties involved in an MCSC campaign can be abstracted to two “conceptual human actors”, the *data collector* and the *device owner* and two “conceptual agent actors”, the *orchestration agent* and the *local agent*. Success of a given campaign depends on these four actors having the “right tools” to perform tasks expected of them. Figure 3 gives an overview (in the form of a use case diagram) of the functions that each actor needs to be able to perform.

In the use case diagram, we can see:

1. A Data collector, which is an initiator of an MCSC campaign, and is expected to perform all configurations, related to the campaign, providing a way for the device owners to participate. In order to achieve this mandate, the data collector requires a campaign management interface providing the ability to (1) *create SRs* and (2) *create DCCs*.
2. An OA requires the ability to (1) *discover the existence of DCCs* and (2) *coordinate and supervise the activity of LAs*.
3. An LA requires the ability to *coordinate sensing tasks* taking place within a sensing device. In order to receive details on the DCC(s) its device is subscribed to, when joining the multiagent platform, an LA must broadcast its presence. This event triggers a notification to an OA, which provides the details of the DCC(s). Using the information obtained from the OA, the LA can configure *sensor alarms* for each DCC, based on its respective SRs.
4. The device owner requires a sensing device interface that provides the ability to (1) *subscribe to DCC(s)*, (2) *unsubscribe from DCC(s)*, (3) *configure sensor data sharing*, (4) *update the device’s node data* and (5) *view a list of plugins (value-added applications) present on the device*.

Task Specification Language

Because not much work has been done at the intersection of MCSC and MASs and, in particular, on using a rule-based approach for sensing coordination, a suitable specification language (e.g., a Domain Specific Language) could not be

TABLE 1. TSL Properties

Property	Required	Allowed values	Requires
samplemode	Yes	“fixed”, “interval” or “continuous”	<i>time</i> if “fixed”, <i>frequency</i> if “interval”
tolerance	No	number or string	-
preprocess	Yes	boolean	preprocessing if true
preprocessing	No*	object	-
time	No*	integer array	-
frequency	No*	integer or string	-

* Conditionally required.

found to fit the use case of the proposed system, especially in the context of supporting complex data processing at the level of a sensing device. Based on this premise, it was decided that a new specification language has to be developed. Two factors were considered when selecting a format for the Task Specification Language (TSL): (1) *ease of reading and writing by a human*, and (2) *ease of parsing and generating by computers*. Given these factors, the *Javascript Object Notation* (JSON) data interchange format was a natural choice for implementing the TSL. JSON is lightweight and language independent but, nevertheless, “facilitates structured data interchange between all programming languages” [15]. It is inspired by the *JavaScript* programming language [16] and is based on two data structures, namely, *objects*, which are collections of name/value pairs, and *ordered lists* (arrays).

Key Features of the TSL

It was decided that, in the TSL, an SR is going to be represented using an object that is made up of a combination of six properties. The properties are *samplemode*, *preprocess*, *preprocessing*, *time*, *frequency* and *tolerance*. Table 1 gives an overview of these properties, their expected values and their interdependencies.

The following information can be found in the Table.

1. The **samplemode** property is a string value that specifies when a sensor is to be read. It may have a value of either *fixed*, *interval* or *continuous*. A value of *fixed* implies that the sensor is to be read at a specific time of the day, every day. When using *fixed*, the *time* property is required. Using a value of *interval* implies that the sensor is to be read at predetermined intervals, starting from the time the alarm is set. This value causes the *frequency* property to be required. A value of *continuous* implies that the sensor is to be read every time its value changes. This value triggers no other required properties.
2. The **tolerance** property is an integer or string expression, representing milliseconds, dictating that the sensor data should be reused if it has been read within the specified time frame. For example, let the tolerance value of a particular SR be set at thirty seconds. If the relevant sensor has been read less than thirty seconds ago, to collect data to be used by any DCC, then that data is going to be reused (rather than a measurement being performed again). This creates a resource saving mechanism that could possibly reduce sensor device battery drain.
3. The **preprocess** property is a boolean value that indicates whether or not the data read from the sensor has to be processed locally, before being sent for aggregation. A value of *false* indicates that no processing is to be done and the data is sent as soon as it is read. A value of *true* indicates that local processing is to be done and, consequently, causes the *preprocessing* property to be required. If no value is specified, *false* is assumed.
4. The **preprocessing** property is where the rules for applying transformations to sensed data are represented. It is a JSON object that contains two sub-properties, *variables* (an object) and *rules* (an array). The *variables* object is where data values, required during local preprocessing, are kept. For example, in the case of the *continuous* sampling mode, the data collector may want to set a “change threshold” that is used to determine if the sensor values have changed enough to warrant being recorded (sent to the Aggregation Server). The *rules* array contains an ordered list of procedures that must be executed in order to transform the sensed data to a required format, calculate needed data, or perform any other operations on the data.

5. The **time** property specifies the time of day a sensor is to be read, accurate to the second.
6. The **frequency** property specifies how often a sensor is to be read, in milliseconds.

Preprocessing Rules

Let us now discuss in some details the form of preprocessing rules in the current version of the TSL. Here, **Arithmetic** rules are essentially mathematical expressions that dictate what operations to perform on sensor values, if some arbitrary condition is met. Listing 1 gives an example of an arithmetic rule. In this example, after the expression is evaluated, the result is stored into a variable called *arithmeticOpOne*. The rule, however, is only executed if the condition is met that the value at index zero of the the sensor data values is greater than the value specified by the *threshvar* variable. The *threshvar* variable could be one set in the variables object or the result of a previously executed rule. The expression is evaluated and the result should be the square root of the sum of the first two values of the sensor data values array.

The **Human input rules** (HIRs) are the mechanisms that allow a device owner to have direct input into the sensing process. HIRs, similarly to the *arithmetic* rules, only run if some arbitrary condition is met. Listing 2 shows an example of a HIR. The rule is always executed, when data is required from the relevant sensor, as the condition for execution always evaluates to true.

The **Query** rules run select queries against the database of values stored on the sensing device, if some arbitrary condition is met, and stores the resulting value to a variable. They are, therefore, best suited to extracting single valued results from the database. It should be noted that the queries are restricted to rows where the sensor identifier is equal to the identifier of the sensor for which the rule is written. The query clauses also support sensor values and variables placeholders. Listing 3 gives an example of a query rule.

The **Store** rules indicate that the result of the rule should be saved to the database, if some arbitrary condition is met, to be used for future querying and possibly for use by plugin applications. If no expression is given, the entire value array of the sensor data is stored to the database. Listing 4 gives an example of a store rule.

Lastly, the **return** rules indicate that, if some arbitrary condition is met, the result of executing rule should be sent for aggregation. If the return type is *string*, variable replacement is performed and the string returned as is. If it is *eval*, however, the expression will be evaluated mathematically and the result returned. If no expression is given, the entire value array of the sensor data is returned. It should be noted that without a return rule, SRs that have the *preprocess* property set to *true* will not send any data for aggregation. Listing 5 gives an example of a return rule.

Listing 1. An example of an *arithmetic* rule.

```

1  {
2    ...
3    {
4      "type": "arithmetic",
5      "id": "arithmeticOpOne",
6      "if": "[0] > {threshVar}",
7      "expr": "SQRT([0] + [1])"
8    }
9    ...
10 }

```

Listing 2. An example of an HIR.

```

1  {
2    ...
3    {
4      "type": 'human_input',
5      "id": 'hirOne',
6      "if": "true",
7      "datatype": "string",
8      "message": "Is the sky cloudy?"
9    }
10   ...
11 }

```

Listing 3. An example of a *query* rule.

```
1 {
2   ...
3   {
4     "type": "query",
5     "id": "query0p0ne",
6     "if": "[0] > {threshVar}",
7     "select": "SUM(column1) as query0p0ne",
8     "where": "column1 != column2",
9     "groupby": "column3",
10    "having": "search_condition",
11    "orderby": "column3",
12    "limit": "100"
13  }
14  ...
15 }
```

Listing 4. An example of a *store* rule.

```
1 {
2   ...
3   {
4     "type": "store",
5     "if": "[0] > {threshVar}",
6     "expr": "[0]"
7   }
8   ...
9 }
```

Listing 5. An example of a *return* rule.

```
1 {
2   ...
3   {
4     "type": "return",
5     "if": "[0] > {threshVar}",
6     "returntype": "eval",
7     "expr": "[0]"
8   }
9   ...
10 }
```

Technology Stack

The management interface utilizes the Linux-Nginx-MySQL-Python (LEMP) technology stack together with the MongoDB document database. The flavor and version of Linux used for the server is Ubuntu Server 16.04.2 LTS [17]. Powering the management interface, on the Ubuntu server, is a combination of the Nginx web server [18] and the uWSGI application container [19], while Python was selected to implement the web application. More specifically, the Python based Flask web application micro-framework [20] has been used. In the case of data storage, a hybrid approach was used. MySQL [21] was used to store data explicitly related to user (data collector) management, while MongoDB [22] was used as a document storage for all MCSC campaign-related data.

The framework behind the agent platform subsystem is the Java Agent DEvelopment Environment (JADE) [23]. In the split container mode of JADE, a peripheral container, in which an agent lives, is split into two constituent containers, a *front-end* container and a *back end* container. The front-end container exists on the peripheral machine while the back-end container exists on the main container's machine. Communication between the front-end and back-end is achieved via *Transmission Control Protocol* (TCP) sockets. The split container execution mode was especially designed for mobile environments, in which network connectivity is intermittent and most ports are closed to incoming traffic. Split containers are, therefore, a natural choice for the LAs that are domiciled on sensing devices in an MCSC scenario. To reduce the time needed to deploy an always online, fault tolerant agent platform using JADE, the *PhaseMetrics.io* platform [24] was used. PhaseMetrics.io is a cloud based platform-as-a-service (PaaS) offering

that allows developers to launch JADE based multiagent systems with just a few clicks. Nevertheless, we realize that the selection of split-container approach is not without potential problems; *i.e.*, caused by lack of Internet. Therefore, we will investigate this issue further (see, also, Section e)).

The current iteration of the system supports only sensing devices running the Android operating system [25]. Android is a Linux based operating system, designed especially for smartphones, used on various consumer electronic devices. The implemented system supports devices running Android versions 4.4 (KitKat) and above. Android applications are developed using the Android Software Development Kit (SDK). The Android SDK is a Java based environment, and, as such, the core of Android applications are developed in the Java. Native code in languages such as C and C++ is also supported. Given that the applications are developed using Java, it was easy to integrate the JADE based agent platform to run the LAs on the sensing device. To equip LAs with the ability to parse mathematical and boolean expressions in SRs represented in the TSL, EvalEx [26], a Java based expression evaluator, has been used.

RELATED WORK

Not many attempts have been made at merging the MCSC paradigm with that of MASs. In fact, a search across major publishers in the field has only uncovered one such attempt. Here, [27] introduced a system of mobile agents for the MCSC, where campaigns are *role-based* implementations of mobile agents. In this system, agents are responsible for tasks including collecting data, analyzing data and sharing data in campaigns. These tasks are considered to be the *roles* of the agents. The agents are equipped with the ability to migrate to and from participating devices, and take into account the available resources of the specific device, such as battery level, in carrying out their assigned tasks. Agents in the system also try to match participants' privacy requirements to campaign requirements.

Though the work of [27] is also based on combining MCSC with MASs, our approaches are fundamentally different. Whereas our approach uses a single MAS to coordinate collaboration between arbitrary campaign operators and data contributors, cited work implements each campaign as a separate MAS composed of role-based agents migrating to and from participating devices to perform specialized tasks. The "campaigner", who is the initiator of a campaign, is required to actively monitor the availability of potential contributors in the system and inject agents, as needed, to service the available participants. This increases the man power needed to monitor campaigns and is in contrast to our approach, where campaigns are created, open calls for participation are published, and the campaign initiator can then step away and let the campaign manage itself.

Additionally, the agent migration approach, used in [27], was consciously avoided in our system design. Allowing arbitrary agents to migrate to participant devices is a huge privacy and security risk for participants, that may deter their participation, as the intentions of these agents can never be guaranteed. Our approach recommends a single agent, permanently domiciled on the sensing device, over which the campaign operator has no control, except for configuring how existing sensors are to be read. Overall, the work reported in [27] showed that there is shared interest, in the research community, in combining MCSC and MASs. Their architectural approach, however, is not well suited to the aims of our work.

PRELIMINARY TESTS OF THE DEVELOPED SYSTEM

We will now discuss the use case scenarios that were introduced earlier, of campaign initiators, Alice, Bob and Eve, and how they would go about launching their MCSC campaigns using the developed system. The focus will be on the user interfaces, provided in accordance with the use cases, given in Figure 3, for both the data collector and the device owner. To begin, Figure 4 shows the interface used to create SRs. The area for entering the SR is supported by syntax highlighting and auto-indentation to make editing easier.

After creating SRs, campaign initiators can now attach them to DCCs. An example of a DCC, with an SR referenced, via selecting a checkbox near its name, is shown in Figure 5. When creating a DCC, the initiators may enter a description that would be shown to potential data contributors, when they search for DCCs to subscribe to. Also, as seen in Figure 5, when a DCC is created a unique QR code is generated.

Once created, DCCs are discoverable by OAs and device owners can subscribe their devices to the DCCs using the developed Android application. Subscription can take place either by using the search interface provided in the application (Figure 6a) or by scanning a QR code (Figure 6b). The search term, entered by the application user, is matched against both the titles of DCCs and their descriptions. The QR code, generated for a DCC, encodes the DCC's

Management Interface Subsystem admin ▾

[Home](#) / [Management Dashboard](#) / [Agent Platform Sensor Rules](#) / [New Rule](#)

New Sensor Rule

Name*

Sensor ID*

Rule*

Tags

FIGURE 4. Interface for creating SRs

Management Interface Subsystem admin ▾


[Home](#) / [Management Dashboard](#) / [Agent Platform Data Collection Channels](#) / [Edit Channel](#)

Edit Channel

Name*

Description*

This channel collects data on a device owner's application usage patterns.
 Required sensors:
 - apps_running



Rules

- AUA Apps Running
- Health Tracking: Step Counter

FIGURE 5. Interface for editing DCCs

unique system identifier and it is this identifier that is used to subscribe the device, when the code is scanned. Both methods of subscribing to a DCC result in an entry to the list of subscribed DCCs (Figure 6c).

In addition to facilitating the subscription process, the Android application provides device owners with features that allow setting non-identifying demographical data (Figure 7a), configuring which sensors to share data from (Figure 7b), and viewing/accessing plugins installed on the device (Figure 7c). With regards to the demographical data, the device owner is allowed to set their gender and age range, while, in the case of sensors, they are able to select hardware or software sensors.

When data aggregation starts, campaign initiators are able to access the data entries via the data pages of their

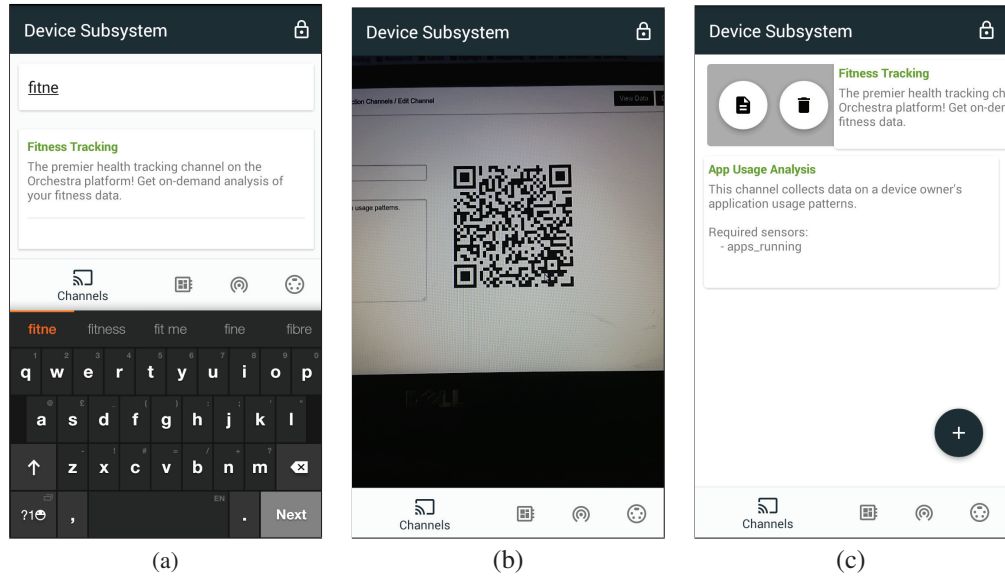


FIGURE 6. Interfaces for subscribing to DCCs

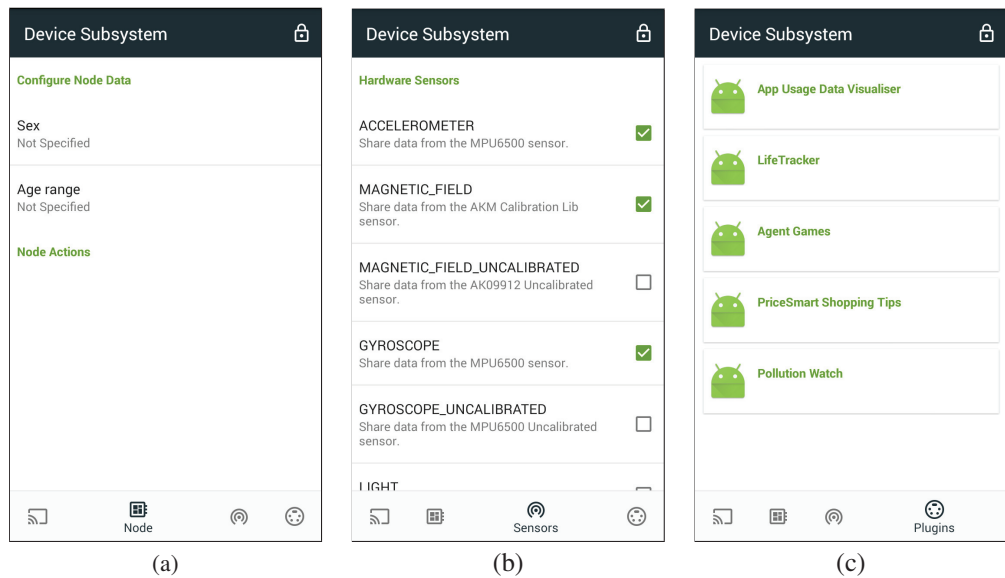


FIGURE 7. Additional interfaces provided by the Android application.

respective DCCs. These pages are accessible via the interface used to edit the DCCs and, for each data entry, indicates the associated sensor, the date that the sensor reading was made (accurate to the second) and the values that were read. Figure 8 shows how this page would look like in a situation when an initiator creates an SR to detect how many apps are running on a device at a given instance. Clicking on any of the entries would output the JSON representation of the sensor reading for that entry (Figure 9). In summary, while the developed system has not been used in an actual campaign, we have shown that, using the provided interfaces, data collectors and device owners are able to execute all required tasks, based on the stipulated use cases.

Channel Data

1292 entries in document store.

#	Sensor	Date	Values	Actions
1	apps_running	2017-06-22 05:56:51	8	view
2	apps_running	2017-06-22 05:57:52	8	view
3	apps_running	2017-06-22 05:58:54	9	view
4	apps_running	2017-06-22 05:59:56	9	view
5	apps_running	2017-06-22 06:00:59	10	view
6	apps_running	2017-06-22 06:02:00	9	view
7	apps_running	2017-06-22 06:02:51	9	view
8	apps_running	2017-06-22 06:03:52	9	view
9	apps_running	2017-06-22 06:04:54	9	view
10	apps_running	2017-06-22 06:05:56	9	view

FIGURE 8. Interface for viewing DCC data

```

1 // 20170713052119
2 // http://orchestra.phasemetrics.io/dashchannels/entries/594b5c23f3ba7e2d2349fced
3
4 {
5   "_id": {
6     "$oid": "594b5c23f3ba7e2d2349fced"
7   },
8   "accuracy": 0,
9   "timestamp": 1498111011,
10  "channelId": "5925faad59798005d71d0bb3",
11  "sensorId": "apps_running",
12  "returnType": "rule",
13  "location": [
14    52.1708472,
15    21.0160745,
16    153,
17    26
18  ],
19  "values": [
20    8
21  ]
22 }

```

FIGURE 9. Sample JSON output for a sensor reading entry

DISCUSSION

Two deficiencies that became apparent, while evaluating the system, were: lack of branching and looping language structures, and structures that would enable device-owner-initiated interactions, in the TSL. Having branching and looping structures in the TSL would be beneficial when there is a need for data collectors to represent complex workflows. For example, in the campaign run by Alice, if SRs supported loops, instead of having one place of interest per SR, there could be multiple places and rules in the *preprocessing* object, and they could each be iterated over to determine if any are close by. In the case of the campaign run by Eve, if SRs supported specifying device-owner-initiated interactions, instead of waiting for the specified times to give feedback on wait times to access public transportation, data contributors could initiate the process, if they happen to have time to deal with it.

Supporting a more complex, expressive, TSL, however, has also its disadvantages. The more complex the TSL becomes, the more skilled the administrators (rule writers) have to be. The TSL will, at some point, approach the difficulty of more traditional high level scripting languages and will, thus, require administrators with skills similar to those of traditional programmers. This point brings us to another limitation of the implemented solution: there is no

simple, graphical, rule building tool, to act as a layer of abstraction between the data collector and the deeply technical aspects of the TSL. Not having a graphical rule builder raises the barrier of adoption as potential data collectors would have to at least have working proficiency with the JSON data interchange format.

Challenges were also encountered during the implementation of the system. The first challenge was that of hosting the OAs in the cloud. Though the PhaseMetrics.io platform provides an intuitive way of launching the critical infrastructure required to deploy the agent platform, it does not provide a way to spawn custom agents on the machine that hosts the main container. This meant that there was no way, by default, to deploy the OAs, since they are domiciled in the agent platform subsystem, and, thus, cannot be placed with the management interface subsystem or the sensing device subsystem. To overcome this challenge, the code for the OAs were developed in isolation from the rest of the system and deployed to a Linux-based cloud server, specifically configured for this purpose. Once started, the container in which the OAs live establishes a connection to the main container via TCP/IP networking. This distributed architecture is supported by JADE and stays true to the nature of the MASs.

Another challenge was in relation to the LA name clashes that prevented devices from reconnecting to the network after the agent would have died. In situations where an LA dies, a “ghost” of the agent remains in the system as the back-end of its split container is left running. Examples of this includes when the device loses power abruptly or the application somehow crashes. This *ghost agent* prevents the LA from reconnecting to the agent platform, as agents with duplicate names are not allowed. To overcome this challenge, a transient agent was introduced to the boot process of an LA. When an LA wishes to start, an *initiator agent* is first launched. This *initiator agent* connects to the agent platform, with its name being a combination of the LA’s name and a random string. The *initiator agent* checks if a *ghost agent* is present in the system and, if present, makes a request to the agent platform to have it killed. After performing this task, the *initiator agent* removes itself from the agent platform and triggers the boot process of the LA. This method has proved effective and LAs are now able to avoid duplicate name errors on joining the agent platform. Nevertheless, this issue will be returned to in the future.

CONCLUSIONS AND FUTURE WORK

In this paper, we have explored the development of a generic, adaptive, system to support arbitrary mobile crowd-sensing and computing (MCSC) scenarios. The focus was on three processes, namely, *data gathering and integration*, *local preprocessing of data*, and *software defined sensing*, and the concepts of *value-added applications* and sensing coordination using a *task specification language* (TSL). We proposed, within the context of these processes and concepts, a system that combines the MCSC paradigm with that of multiagent systems (MASs). The objectives of the research were to design an architecture for such system, design a (domain specific) task specification language, used to instruct intelligent agents on how to coordinate sensor reading, and implement a prototype that demonstrates the feasibility of the proposed designs.

We presented a comprehensive description of both the architecture and the TSL, and the technology stack used during the implementation. Further to this, a discussion of related work taking place at the intersection of MCSC and MASs was presented. Finally, the developed prototype was evaluated against a set of use cases deemed essential to the success of such a system, and a discussion was presented on the current limitations of the system and some challenges that were encountered during the implementation. It is our conclusion that the prototype satisfies the objectives of the research and truly represents a generic, and adaptive, platform for MCSC. With further development, the system could reach a point of maturity where it will have significant impact on the way mobile data is collected, used for knowledge discovery, and shared between different applications, organizations and the wider public.

Future work can be divided into three categories, namely, theoretical, language specification and software engineering. In the case of theoretical work, it would be beneficial, to the mobile computing field, if work is done to establish a sound theoretical framework for the discussion of MAS-based MCSC systems. This would involve performing a thorough literature review in both fields and extracting the significant concepts that can be used to classify and characterize such a system. In addition, exploring open challenges in the field of MCSC, as outlined in [13] would prove a worthy endeavour. With regards to the language specification category, work would focus on improving the expressiveness of the domain specific TSL. The JSON-based TSL lacked support for branching and looping structures in the rules used to preprocess sensor data. This shortcoming should be addressed. The question of other developments of the TSL, however, needs to be approached judiciously, so as not to over-complicate the language, thus reducing its uptake.

In the software engineering category, future work would focus on completing the functionality of the platform, *e.g.*, making published MCSC campaigns visible through a public page of the web application used to manage them,

and operational testing to examine the scalability of the system. The platform must be able to handle large number of MCSC campaigns, each with large numbers of sensing devices, and must be evaluated as such. Operational testing would include evaluating simulations of data transfers at massive scales, concurrently across multiple MCSC scenarios.

ACKNOWLEDGEMENTS

Work presented here is partially supported by the National Scientific Fund of Bulgaria under the grant DFNI-DN 02/10 “New Instruments for Knowledge Discovery from Data, and their Modeling” and by the Polish-Bulgarian collaborative grant “Parallel and Distributed Computing Practices.”

REFERENCES

- [1] J. Dutta, F. Gazi, S. Roy, and C. Chowdhury, “Airsense: Opportunistic crowd-sensing based air quality monitoring system for smart city,” in *2016 IEEE SENSORS* (2016), pp. 976–978.
- [2] N. Rapousis and M. Papadopouli, “Performance analysis of a user-centric crowd-sensing water quality assessment system,” in *2016 International Workshop on Cyber-physical Systems for Smart Water Networks (CySWater)* (2016), pp. 13–18.
- [3] Mediated Spaces, Inc., The wildlab, <http://www.thewildlab.org> (accessed 2017-06-07).
- [4] Networked Organisms, Project noah, <http://www.projectnoah.org> (accessed 2017-06-07).
- [5] F. Seraj, N. Meratnia, and P. J. M. Havinga, “Rovi: Continuous transport infrastructure monitoring framework for preventive maintenance,” in *2017 IEEE International Conference on Pervasive Computing and Communications (PerCom)* (2017), pp. 217–226.
- [6] A. Ramazani and H. Vahdat-Nejad, *IET Intelligent Transport Systems* 11, 326333 (2017).
- [7] Y. Ding, C. Chen, S. Zhang, B. Guo, Z. Yu, and Y. Wang, “Greenplanner: Planning personalized fuel-efficient driving routes using multi-sourced urban data,” in *2017 IEEE International Conference on Pervasive Computing and Communications (PerCom)* (2017), pp. 207–216.
- [8] B. Guo, H. Chen, Z. Yu, X. Xie, S. Huangfu, and D. Zhang (2015) *IEEE Transactions on Mobile Computing* 14, 2020–2033.
- [9] X. Hu, T. H. S. Chu, H. C. B. Chan, and V. C. M. Leung (2013) *IEEE Transactions on Emerging Topics in Computing* 1, 148–165.
- [10] R. K. Ganti, F. Ye, and H. Lei (2011) *IEEE Communications Magazine* 49(11), 32–39.
- [11] B. Guo, Z. Yu, X. Zhou, and D. Zhang, “From participatory sensing to mobile crowd sensing,” in *2014 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)* (2014), pp. 593–598.
- [12] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M. B. Srivastava, “Participatory sensing,” in *Workshop on World-Sensor-Web (WSW06): Mobile Device Centric Sensor Networks and Applications* (2006), pp. 117–134.
- [13] B. Guo, Z. Wang, Z. Yu, Y. Wang, N. Y. Yen, R. Huang, and X. Zhou (2015) *ACM Comput. Surv.* 48, paper 7, 31p.
- [14] M. Luck, P. McBurney, O. Shehory, and S. Willmott, *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)* (AgentLink, 2005).
- [15] ECMA International, The JSON Data Interchange Format, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf> (October 2013).
- [16] Mozilla Developer Network, Javascript, <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (2017).
- [17] Canonical Group Ltd, Ubuntu Server 16.04.2 LTS, <https://www.ubuntu.com/> (April 2016).
- [18] Nginx, Inc., Nginx web server, <https://nginx.org/en/> (2017).
- [19] Unbit s.a.s, The uWSGI project - uWSGI 2.0 documentation, <https://uwsgi-docs.readthedocs.io/en/latest/> (2017).
- [20] A. Ronacher, Flask (A Python Microframework), Version 0.12, <http://flask.pocoo.org/> (December 2016).
- [21] Oracle Corporation, Mysql, <https://www.mysql.com/> (2017).
- [22] MongoDB Inc., MongoDB, <https://www.mongodb.com/> (2017).

- [23] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, EXP – In Search of Innovation 3, 6–19 (2003).
- [24] PhaseMetrics, Inc., Phasemetrics.io – a suite of tools for deploying and managing multiagent-based distributed systems, <http://phasemetrics.io> (2017).
- [25] Google, Android, <https://www.android.com/> (2017).
- [26] U. Klimaschewski, Evalex – java expression evaluator, <https://github.com/uklimaschewski/EvalEx> (2017).
- [27] T. Leppänen, J. Álvarez Lacasia, Y. Tobe, K. Sezaki, and J. Riekkı (2017) *Autonomous Agents and Multi-Agent Systems* **31**, 1–35.