Standardized Scalable Relocatable Context-Aware Middleware for Mobile aPplications (SCAMMP)

Fatima Abdallah

Faculty of Sciences
Lebanese University
Beirut, Lebanon
Email: f.3abdallah@gmail.com

Hassan Sbeity and Ahmad Fadlallah

Faculty of Computer Studies
Arab Open University
Beirut, Lebanon
Email: {hsbeity,afadlallah}@aou.edu.lb

***Abstract*—The penetration of handheld devices (especially smartphones) is predicted to be over one billion in the next five years. These devices are increasingly equipped with new sensors offering a great potential for developing context-aware mobile applications that can enhance user experience. Unfortunately, the data provided by these sensors are of low-level (raw data) and diverse, ranging from physical to virtual. This makes embedding contextual information into mobile applications a difficult task. Presenting these raw sensors' data in a unified format, augmenting them into useful high-level context information and offering them through a well formalized standard middleware service can make this task easier. In this work, we present the architecture of a middleware platform that provides an open standard interface offering high-level information to the application layer. This platform maintains user context information in a finite state machine through state engines. State engines that represent user states can be added and removed any time and hence the openness of the platform originates. The platform uses layered approach and is composed of two relocatable layers: data acquisition-augmentation (pre-processing) layer and decision layer. A case study was performed to validate the functionality of the platform.**

Keywords–Context Awareness; Middleware; Mobile Applications.

I. INTRODUCTION

Most today smart devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy. Every smartphone nowadays is equipped with a set of small sensors. These sensors can be hardware-based embedded in the smartphone (e.g., acceleration sensor) or software-based that derives their data from one or more hardware-based sensors (e.g., linear acceleration sensor). A third category of sensors could be introduced, which is the logical sensors (e.g., calendar events).

Providing mobile applications with high level sensor information can enhance the efficiency of these applications toward power saving and user experience. Examples are many. For instance when the user is traveling, application that needs synchronization with cloud services (data upload/download through mobile network), can postpone these jobs until the user is at home or at work in order to save battery power even if the mobile network provides a high bandwidth data connection over LTE (Long Term Evolution) for instance. Because once the battery of the mobile is drained, recharging the phone is difficult while traveling. Another example is making the phone silent when the user is sleeping, which enhances the user experience. A power friendly operating system process scheduler can swap processes from memory to persistent storage while taking into consideration the user state. A reminder application can notify the user events not only based on time and dates but also based on his location and according to what he/she is doing. For instance, an alarm can be set based on date time and user states; one can choose ringing when sleeping only, awake only, or both. Traditionally, applications have input from the user, persistent storage, and recently from the network via Remote Procedure Calls (RPC) for message passing. But offering meaningful user context information originally gathered from different physical and virtual sensors, provides a new input source that, if standardized, will provide a new brand of applications. Furthermore, the history of these high-level user context information can be used as a user signature to authenticate the user to his/her device.

In this work, we present the architecture of a middleware platform that provides an open standard interface offering high-level information to the application layer. This platform maintains user context information in a finite state machine through state engines. State engines, that represent user states, can be added and removed any time, and hence the openness of the platform originates. The platform uses layered approach and is composed of two relocatable layers: data acquisition-augmentation (pre-processing) layer and decision layer. We also present a case study that decides about the current user location, its purpose is to test the validity of the middleware by mapping the different components of the application to the different SCAMMP modules.

The rest of this document is organized as follows: Section II presents the architecture of the middleware platform. Section III describes the different blocks of the decision layer. Section IV explains the modules of the Data Acquisition-augmentation Layer. Section VI explores a set of related work and compares them with our work. Section V presents a case study as an evaluation of the functionalities of the different SCAMMP components. Finally, Section VII concludes the paper and presents the future work.

II. ARCHITECTURE

Fig. 1 depicts the general architecture of the middleware platform. The application layer represents any mobile ap-

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-353-7

plication that might embed context-aware information. The Decision Layer located at the top of the middleware platform (SCAMMP) provides the application layer with an Application Programming Interface (API) to access high-level context-aware information. This information is stored in a finite state machine and represents in a real time fashion the different user states. Hence, any application can easily integrate user context information. At the bottom is the Acquisition Layer, its main role is to represent the data captured from different sensors in a unified XML format.

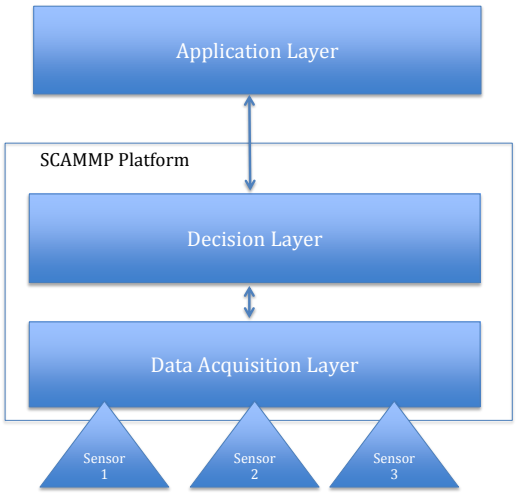

Figure 1. SCAMMP Architecture.

Each layer provides services to the layer above through a well-defined set of commands (protocol). At each layer, modules can be added and removed dynamically, hence guaranteeing the openness and the scalability of the platform. The communication between the different layers is accomplished as follows: The lower layer sends notification (using push mechanism) to the upper layer each time a sensor notification is received signaling the availability of valid data. If the upper layer is interested, it issues a request asking for the new update using a predefined protocol. Moreover, at any time the upper layer can issue a command to the lower layer asking for data updates. The separation of the system into different layers offers a great flexibility for layer hosting and hence the relocatabilty of the platform is originating.

III. DECISION LAYER

The main task of the decision layer is to maintain a finite state machine that reflects the different user states in a soft real time. Fig. 2 depicts the different components that build up this layer. All components should reside in the same address space and hence, intra-communication can be done through shared variables.

The API component represents the interface to the application layer and the controller component provides an interface to the lower layer, namely the data acquisition layer. All remaining components have no direct access outside this layer. This layer can be hosted on the mobile device or on the cloud; it can be relocated depending on the available bandwidth. The history of the user states can also be uploaded on the cloud.

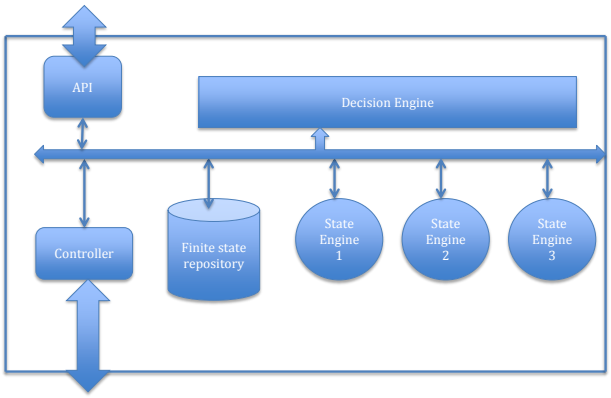

Figure 2. The Architecture of the Decision Layer.

A. API

The API component is the only interface provided to the application layer to communicate with SCAMMP. A set of pre-defined commands (protocol) are used as a communication mean between the application layer and the SCAMMP. In fact, the current and the history of the different user states will be made available to the application layer. For instance, currently the user is at home (state) and is sleeping (attribute). The API component has access to the state repository where the current and the history of the different user states can be found. The communication protocol can be simple, such as the HTTP, where any application at the application layer can send a request and receive response. There are two types of requests that can be sent to the API component, one that requests a list of the available user states and their attributes (names and descriptions) and one that requests the current or the history of the different user states for a specific period of time.

B. Decision engine

The decision engine is responsible for updating the different user states (along with the corresponding attributes). The kernel of the decision engine is based on a mathematical model that decides about the current user state. The decision is based on input from the different state engines. Each time a state engine makes an update, the decision engine is informed in order to recheck the user state and eventually update it. The outcome of this engine will be made available to the application layer through the API component.

C. Finite state repository

The finite state repository is a storage system. It contains three types of data, two of them are available to the application layer through the API component and the third one is only for internal use.

The first data is an XML entry list that contains an entry of every registered state engine. The second data is an XML entry list that stores the current and the history of the different user states. The third data is also an XML entry list that contains the output of every state engine. It is only for internal use and is available for the decision engine. Because the history of the user state could be huge after a while, it can be archived and

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-353-7

uploaded to the cloud, while still be available to the application layer.

D. State engine

Every user state can have many attributes, for instance, home is a user state and sleeping is its attribute. Every state engine is attached to one or more sensor agents of the lower layer. The finite state engines take input from the sensor agents through the controller, and produce output in the finite state repository. The data produced by the finite state engine are only for internal use, namely the decision engine use them as input. The main task of the state engine is to calculate the certainty of a certain user state and it is left to the decision layer to decide which is the current state of the user.

Each time a sensor agent sends a notification to announce the availability of new sensor data, the corresponding finite state engine will be informed via the controller. It is up to the state engine to decide whether to request or not the data. A new finite state engine should be introduced to the controller in order to be registered. During the registration of a new state engine, a list of the corresponding sensor engines of the lower layer needs to be specified.

E. Controller

The controller's main role is to maintain the communication with the lower layer and hence isolating the decision layer form the data acquisition layer. It receives notifications form the sensor agents of the lower layer and forwards them to the corresponding state engines. It sends requests to the lower layer on behalf of the different finite state engines and forwards the response to the corresponding finite state engines. The second role is to maintain a list of the active finite state engines. Each time a new state engine is introduced, it has to be registered at the controller.

IV. Data Acquisition-augmentation Layer

The main role of the Data Acquisition-augmentation layer (see Figure 3) is to provide a unified format of the data collected from different sensors.

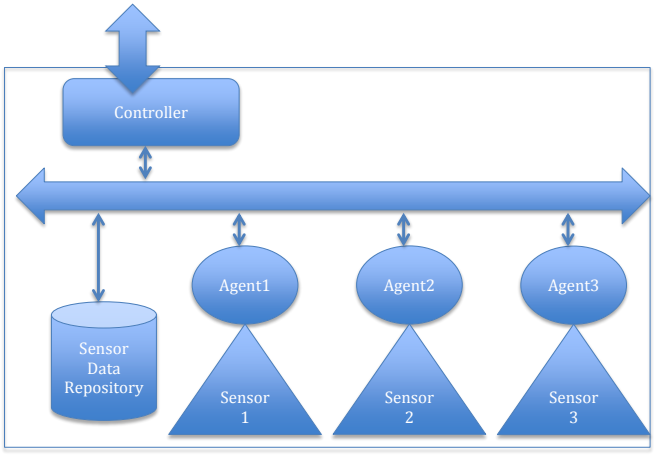

Figure 3. Architecture of the Acquisition layer.

Every sensor whether physical or virtual will be attached to a single dedicated agent. Once a sensor has produced a new data, the corresponding engine will decide according to a certain threshold whether to forward a notification to the upper layer or not; if yes, the agent will collect the data and store it in the data repository in a unified XML schema (see Figure 4).

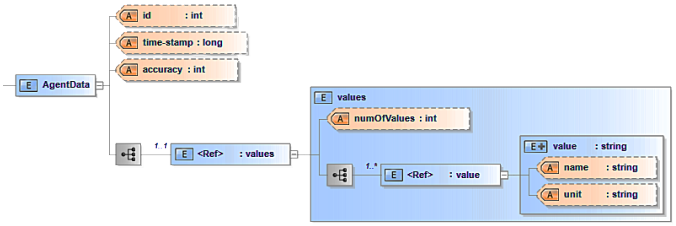

Figure 4. Unified Agent Data XML schema.

These data will be made available to the upper layer through the controller.

A. Sensors

Most smartphones nowadays are equipped with many small sensors. As previously mentioned, some of these sensors are hardware-based and some are software-based. Hardware-based sensors are physical components built into handsets or tablet devices. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change. Software-based sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors. The linear acceleration sensor and the gravity sensor are examples of software-based sensors. Another category of sensors is logical sensors, such as the tweets and the calendar events.

B. Agents

Most popular mobile operating system (OS) provides sensor framework as an API. For instance, Android-powered mobile devices provide raw sensor data by using the Android sensor framework. The sensor framework is part of the android hardware package and includes many classes and interfaces (SensorManager, Sensor, SensorEvent, SensorEventListener, etc.). The agents encapsulate the OS framework to provide a homogeneous sensor data representation. A unified XML schema is used by all agents to represent the captured sensor data. The main role of the agent is to convert the raw sensor data into a unified XML format (see Figure 4). Examples are illustrated for the three categories of sensors (physical, virtual and logical) in Figs. 5 (for the accelerometer sensor), 6 (for the Battery sensor), and 7 (for the calendar sensor). This is done using a threshold that is based on the difference of two consecutive carried data. For every sensor (whether physical, virtual, or logical) there will be a dedicated agent. The internal implementation of the agent is sensor-dependent.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-353-7

```
<AgentData id="1"
           time-stamp="10000"
           accuracy="1">
    <values numOfValues="3">
        <value name="x"
               unit="m/s^2">9.45</value>
        <value name="y"
               unit="m/s^2">1.34</value>
        <value name="z"
               unit="m/s^2">2.7</value>
    </values>
</AgentData>
```

Figure 5. Agent captured data of the Accelerometer sensor.

```
<AgentData id="2"
           time-stamp="10990"
           accuracy="1">

    <values numOfValues="2">
        <value name="Level"
               unit="unitless">88</value>
        <value name="Status"
               unit="unitless">discharging</value>
    </values>

</AgentData>
```

Figure 6. Agent captured data of the Battery sensor.

```
<AgentData id="3"
           time-stamp="00"
           accuracy="1">
    <values numOfValues="3">
        <value name="eventName"
               unit="unitless">Meet Manager</value>
        <value name="from"
               unit="hh:mm">11:30</value>
        <value name="to"
               unit="hh:mm">12:10</value>
        <value name="date"
               unit="dd/mm/yyyy">10/12/2014</value>
        <value name="reminderTime"
               unit="minutes">60</value>
    </values>
</AgentData>
```

Figure 7. Agent captured data of the Calendar sensor.

Each time a new agent is created, it has to be registered in the controller repository using a unified XML format (see Figure 8). Fig. 9 depicts the content of an agent's registration file containing three different agents.

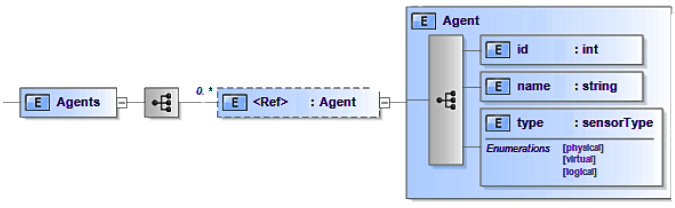

Figure 8. Agent directory XML schema.

```
<Agents>
    <Agent>
        <id>1</id>
        <name>Accelerometer</name>
        <type>physical</type>
    </Agent>
    <Agent>
        <id>2</id>
        <name>Camera</name>
        <type>logical</type>
    </Agent>
    <Agent>
        <id>3</id>
        <name>Calender</name>
        <type>virtual</type>
    </Agent>
</Agents>
```

Figure 9. Agents registration file.

C. Controller

The controller is the interface of the data acquisition-augmentation layer to the decision layer. The controller roles are to:

- Maintain a repository that has an entry for every registered agent using the unified XML format (see Figure 8)
- Receive notifications from the agents and issue simple commands to the agent, such as switch the sensor data acquisitions on and off.
- Forward notifications to the upper layer once received from the agents.
- Have read access to the repository that holds the data stored using a unified format (see Figure 4) collected from the different agents.

V. Case Study

The main goal of SCAMMP is to provide an open standard middleware that offers user context-aware information through an API to the application layer. Hence, any application that wishes to integrate context-aware information can use this API . In order to evaluate SCAMMP, we consider one case study that decides about the current user's location. By design, SCAMMP is intended to host decision logic . Thus, we decided to integrate user-location logic in order to evaluate SCAMMP by mapping the different components of the application in the different SCAMMP modules. It is important to mention that this mapping is done statically in order to evaluate the functionality of SCAMMP; The intra- and inter- communication of the different SCAMMP entities is simulated (done manually since the system is not fully implemented yet).

A. Data Acquisition-augmentation layer mapping

We define three agents required to determine the user's location. These agents embody the following sensors: Location

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-353-7

(hardware/software sensor), network connection (software sensor) and calendar (logical sensor) sensors. They are considered as input for the "Location State Engine". These agents convert the raw data generated by the embodied sensors into a unified XML data as per the schema defined in Fig. 4.

Location Agent: Both Android and IOS operating systems offer a location framework that determines the location of the device. It is a software-based sensor that uses the GPS, cell tower information, and connected Wifi network to detect the user's location. It returns the location using the attributes: longitude, latitude, altitude, accuracy in meters, and time. Fig. 10 is a sample data produced by the Location Agent.

```
<?xml version="1.0" encoding="utf-8"?>
<AgentData id="4"
           time-stamp="1000"
           accuracy="20">
    <values numOfValues="3">
        <value name="longitude" unit="degree">35.528873</value>
        <value name="latitude"  unit="degree">33.8662331</value>
        <value name="altitude"  unit="meter">0</value>
    </values>
</AgentData>
```

Figure 10. Location Agent Data.

Network Connection Agent: This agent determines the type of network the user is currently connected to (e.g., WiFi and Mobile Data network). This information can be obtained from the "Connectivity Manager" in the mobile operating system. The agent's role is to send a notification whenever the user switches from one type of connection to another. The returned data includes: connection type, connection SSID for Wifi networks, and the set of cell towers the device is connected to. Fig. 11 represents real sample data for a Wifi network connection with SSID 'Alfa'.

```
<?xml version="1.0" encoding="utf-8"?>
<AgentData id="5"
           time-stamp="1000"
           accuracy="100">
    <values numOfValues="2">
        <value name="type" unit="unit-less">Wifi</value>
        <value name="ssid" unit="int">Alfa</value>
    </values>
</AgentData>
```

Figure 11. Network Connection Agent Data.

Calendar Agent: This is a logical agent that can be either local or hosted on the cloud. The information collected from the embodied sensor can be used (at the decision layer) to raise the certainty of the location obtained from other agents. Fig. 12 is a sample real data presenting a calendar event named 'Meet Manager'.

B. Decision layer mapping

The agents presented above are attached to a single engine called "Location State Engine" at the decision layer. This engine's kernel can determine user's location (Home, Work, or elsewhere) using Relational Markov Model [7], and K-Nearest Neighbors (KNN) Algorithm [8].

```
<?xml version="1.0" encoding="utf-8"?>
<AgentData id="3"
           time-stamp="93262"
           accuracy="100">
    <values numOfValues="5">
        <value unit="unit-less"  name="eventName">Meet Manager</value>
        <value unit="hh:mm"      name="from">11:30</value>
        <value unit="hh:mm"      name="to">12:10</value>
        <value unit="dd/mm/yyyy" name="date">10/12/2014</value>
        <value unit="minutes"    name="reminderTime">60</value>
    </values>
</AgentData>
```

Figure 12. Calendar Agent Data.

C. Simulation

The engine will remain for a period of time collecting the locations that the user frequently visits (learning phase). It associates for each location the active network connection of the device and the time of identifying this location. After collecting data, the location history is analyzed using a simple heuristic to determine the user's home and work locations. This heuristic will work only for users with fixed work location, since it is most likely that a user is at home at night time, and at work in weekdays in the middle hours of the day. To overcome this drawback many works have been done to obtain personal significant places from raw location data using Relational Markov Model, and K-Nearest Neighbors (KNN) Algorithm. We choose to use KNN as a classification method, for this sake we recorded the locations of a mobile holder for 3 days in a frequency of one hour. Each location is classified by one of the labels {0,1,2} corresponding to {Home, Work, Elsewhere} respectively. Table I presents a sample of the collected data.

TABLE I. TRAINING DATA SAMPLE

Date	Time	Longitude	Latitude	Location
4-9-2014	10:06:45	35.5263591	33.8657569	1
4-9-2014	11:05:37	35.5181525	33.8368607	2
4-9-2014	12:45:07	35.5638238	33.8653605	1
4-9-2014	14:45:40	35.5291208	33.8660798	1
4-9-2014	15:03:26	35.528873	33.8662331	1
4-9-2014	16:19:06	35.5146795	33.8498035	0
4-9-2014	17:21:33	35.514577	33.849815	0

During the training phase the "Location State Engine" uses only the Location Agent, it transforms the training data (longitude, latitude) into a matrix. This matrix is the input of the processing phase where the KNN classifier is obtained. As a simulation, we used Matlab [9] to create the classifier. The produced classifier predicted incorrectly 3% of the training data for k=3. The output of the state engine is an XML file dedicated for internal use (see Figure 13), it is used by the decision engine to aggregate outputs from different state engines and provide the final users state through the API.

After setting the classifier, and in order to save battery power, the "Location State Engine" can use the active network connection to decide the user's location without using the Location Agent. Since the device usually connects, the accuracy of the decision is raised by the calendar events. So if an event points that the user created a shopping checklist or have a meeting at a specific hour, the engine could confirm his location in the shop or at work according to the time.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-353-7

```
<StateEngineData id="1"
                 time-stamp="293722"
                 accuracy="80">
    <values numOfValues="1">
        <value name="Location">Home</value>
    </values>
</StateEngineData>
```

Figure 13. State Engine output sample.

VI. Related Work

The multiplicity and diversity of sensors embedded within mobile devices makes context aware applications difficult to develop, so middleware solutions were proposed to provide an abstraction layer between the operating system and applications. In this section, we review some of proposed middleware solutions of the context-aware systems.

Baldauf et al.[1] in their survey over context aware systems, concluded that there is a common layered conceptual framework for most systems: Starting from the low level (Sensors Layer) passing through the Raw Data Retrieval Layer and Preprocessing Layer that raise the level of abstraction of context data. Then the Storage and Management Layer that provides an interface for the Application Layer to obtain what is needed from the collected data. Although most systems have a common architecture, but they differ in the kind of target applications they serve. Some of the architectures gather information for general context-aware applications, such as smart homes, intelligent vehicles, and context aware hospitals. Other systems, including SCAMMP, are specialized for mobile devices.

Henricksen et al. [2] proposed the **PACE** middleware that supports heterogeneity, mobility, traceability and control, and deployment and configuration of new components, which are some of the requirements for context-aware middleware. On the other hand, it doesn't achieve the scalability requirement. This middleware is developed for context-aware systems in general rather than mobile devices. The middleware is divided into 3 layers: *Context Repositories Layer*, *Decision Support Tools Layer*, and *Application Components Layer*. Dey et al. [3] presents a framework that supports the rapid development of general context-aware applications. Using the *Context Widget*, *Interpreters*, *Aggregators*, and *Services*, it separates the context acquisition from the use of context in the application. The **Context Toolkit** was implemented to instantiate the framework, but it is limited in scalability and ease of deployment and configuration. Another generic context-aware framework is **CMF** [4]. It is a scalable context-aware framework that enables processing and exchange of heterogeneous context information. The *Context Source* uses reasoning techniques to integrate data collected from different sensors, and offers them to the *Context Provider*. The framework takes benefits from user profiles stored in the *User Management* component. The sentient object model is proposed by the **CORTEX** [5] project for the development of context-aware applications in mobile ad hoc environments. A sentient object is a mobile intelligent software component that senses the surrounding environment via sensors and other sentient objects. It consists of three parts: *Sensory Capture*, *Context Hierarchy*, and *Inference Engine*. The model was improved more in [6] by adding the reflection capability and the Service Discovery component. It was also tested by building an intelligent vehicle application.

All the referenced approaches are similar to **SCAMMP** in the way they collect data from different sensors, raise its abstraction level, and offer context information for applications. But SCAMMP is a standard, relocatable, and scalable middleware for applications targeting handheld devices. Using the layered approach allows any of the layers to be hosted on cloud. In addition, the collected data can be stored on a remote server to overcome the storage limitation. The scalability of the middleware is obtained by using a unified XML schema to register additional state engines and physical, virtual, or logical sensors. The provided API feeds applications with high level context information in a finite state model that represents the user's state (Home, Work, Traveling, etc.).

VII. Conclusion and Future Work

The main goal of SCAMMP is to provide an open and standard middleware, that offers user context-aware information to the application layer through a well-defined API, which can be accessible by any future application wishing to integrate context-aware information. SCAMMP is open in a way that new user state engines can be added dynamically. It is relocatable allowing, for instance, the decision layer to be hosted on the cloud. It uses standard protocol for inter-layer communication and URI for name spacing. These different aspects (standard, open, and relocatble) distinguish SCAMMP from the currently proposed middlewares. Our current and future works are the detailed design and implementation of SCAMMP, and its evaluation through different case studies.

References

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

[2] K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam, "Middleware for distributed context-aware systems," in On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE. Springer, 2005, pp. 846–863.

[3] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications," Human-computer interaction, vol. 16, no. 2, 2001, pp. 97–166.

[4] H. Van Kranenburg, M. Bargh, S. Iacob, and A. Peddemors, "A context management framework for supporting context-aware distributed applications," Communications Magazine, IEEE, vol. 44, no. 8, 2006, pp. 67–74.

[5] G. Biegel and V. Cahill, "A framework for developing mobile, context-aware applications," in Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on. IEEE, 2004, pp. 361–365.

[6] C.-F. Sørensen et al., "A context-aware middleware for applications in mobile ad hoc environments," in Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing. ACM, 2004, pp. 107–110.

[7] C. Zhou, N. Bhatnagar, S. Shekhar, and L. Terveen, "Mining personally important places from gps tracks," in Data Engineering Workshop, 2007 IEEE 23rd International Conference on. IEEE, 2007, pp. 517–526.

[8] L. Liao, D. J. Patterson, D. Fox, and H. Kautz, "Building personal maps from gps data," Annals of the New York Academy of Sciences, vol. 1093, no. 1, 2006, pp. 249–265.

[9] Matlab the language of technical computing. http://www.mathworks.com/products/matlab/index.html. (2014 (accessed April 10, 2014))

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-353-7