Scalable extensible middleware framework for context-aware mobile applications (SCAMMP)*

Hassan Sbeyti[1†], Mohamad Malli[1], Khalid Al-Tahat[2], Ahmad Fadlallah[1], and Mohamad Youssef[3]
[1]*Faculty of Computer Studies, Arab Open University, Beirut, Lebanon*
{hsbeity, mmalli, afadlallah}@aou.edu.lb
[2]*Faculty of Computer Studies, Arab Open University, Amman, Jordan*
k_tahat@aou.edu.jo
[3]*Faculty of Sciences, Lebanese University, Beirut, Lebanon*
mohamad.h.yousef@gmail.com

Abstract

The number of users of handheld devices will exceed one billion in the coming five years[1]. These devices are increasingly being enhanced with new sensors, which enable the development of context-aware mobile applications. Moreover, mobile applications might share the same contextual information (decision logics) which in return shares data from the same sensors; this introduces high code redundancy. Scalable extensible middleware framework for context-aware mobile applications (SCAMMP) simplifies the aggregation and sharing of raw data from different sensors and the dynamic injection of decision logics in order to generate high level contextual information. It allows mobile applications to share these high level contextual information (decision logic) via a simple Application Programming Interface (API). This paper presents a fully-implemented SCAMMP (on Android platform) with a quantitative performance analysis. The analysis shows that SCAMMP's overhead due to power consumption is around 0%, processing power have peaks less than 14% at certain moments but is zero most of the time, and the memory usage did not exceed 5 MB. It also shows that SCAMMP maintains its scalability after injecting additional decision logics and incorporating more sensors. Furthermore, SCAMMP allows applications to access high-level contextual information by adding only three lines of codes.

Keywords: Context-Awareness, Middleware, Mobile Applications

1 Introduction

The penetration of handheld devices (especially smartphones) is predicted to be over one billion in the next five years [1] [2]. Most of today's handheld devices are equipped with sensors that provide motion, orientation, and other contextual conditions. Thy provide also high precision raw data . In fact, these sensors can be classified as hardware-based (e.g., acceleration sensor) or software-based sensors (e.g., linear acceleration sensor). One can also list a third category of sensors which is the logical sensors (e.g., calendar events).
Allowing any mobile application to learn and dynamically adjust its behavior to the current context (i.e the current state of the user, the current computational environment, and the current physical environment) can enhance the efficiency of these applications towards power saving and user experience. All of these make smart phone applications smarter. Examples are many:

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 7:3 (September 2016), pp. 77-98
*The authors are grateful to Arab Open University- Jordan Branch for supporting this research.
†Corresponding author: Arab Open Univeristy, Omar Bayhoum Str. - Park Sector, Beirut 2058 4518, Beirut, Lebanon, Tel: +961-1392139 (ext: 509), Fax: +961-1392145
[1]Global mobile devices and connections in 2014 grew to 7.4 billions

- When the user is traveling, applications that need to sync with cloud (using expensive mobile data network), can schedule these tasks to be executed at home or at work in order to save battery power.
- Making the phone silent when the user is sleeping enhances the user experience.
- A Power-friendly Operating System process scheduler can swap processes from memory to persistent storage taking into consideration the user's state.
- A reminder application can notify the user's events additionally based on the user's location and according to what he/she is doing. For instance, an alarm can be set based on date, time and current user state; thus, the user can set the alarm to ring when sleeping only, awake only or both.

Embedding contextual information within any mobile application raises the following challenges:

1. Collecting raw data from different sensors that are available in different formats.
2. Developing decision logics that aggregate the different formatted raw data and augment it into useful contextual information.
3. Moreover, different mobile applications might share the same contextual information (decision logics) which in return shares data from the same sensors; this introduces high code redundancy and hence increasing the load on the system resources.

These are the main challenges for integrating contextual information within any mobile application. To address all these challenges, we propose SCAMMP that is an extension of the Standardised Scalable Relocatable Context-Aware Middleware for Mobile Applications research work [3][4]. The current work differs from the previous work by the following features:

1. the previous work presents only the architecture of the middle-ware, the necessary design diagrams and a simulated case study.
2. While in this work: the whole middle-ware platform has been implemented on an android platform.
3. Two case studies were implemented and integrated within SCAMMP
4. Many design issues that were presented in the previous work has been revised and enhanced for performance reason, for instance the communication between the different layers has been radically modified.
5. A quantitative performance analysis was conducted to evaluate the overhead of the SCAMMP on the OS in term of CPU load, memory usage and power consumption.

SCAMMP is composed of two separate layers:

- The Data Acquisition-augmentation Layer (DAL) (pre-processing) that can have access to any sensor (whether hardware, software or logical) using mediating agents (each sensor will be encapsulated using a single agent). This addresses the first and third challenges.
- The Decision Layer (DL) can be used to inject decision logics within so-called state engines (each engine encapsulates a single decision logic that maintains the states of a specific context) and to connect them to specific agents (sensors) of the DAL. Thus, state engines produce high level contextual information using finite state machine. Hence, applications will be able to share this high level context-aware information through a simple API.

SCAMMP architecture is designed to address the above mentioned challenges. It aims to:

- Allow decision logics to share sensor's data, and further sharing decision engine output with applications at the application layer.
- Allow applications on the same device to share in a simple way high level context-aware information and further reduce code redundancy.
- Facilitate the injection of new decision logics, the embedding of future sensors, and the sharing of SCAMMP source code All of this will speed up the testing of new decision logics and will also lead to the extension and sharing of SCAMMP.

Furthermore, we present an evaluation of the overhead (i.e., power consumption, processing power, and memory usage) produced by SCAMMP by injecting decision logics and aggregating data from different sensors.
The rest of this article is organized as follows: section 2 presents the architecture of SCAMMP, the different blocks of the both layers, the data acquisition and the DL. Section 3 presents fully implemented tow decision logic case studies. Section 4 presents a quantitative performance analysis of SCAMMP. Section 5 presents related work and draws a comparaison. Finally, Section 6 concludes the paper and presents future work.

2 Architecture

Figure 1 depicts the detailed architecture of SCAMMP[3]. From the Application layer on the top, any application can share the high-level context-aware information provided by SCAMMP. The DL which is located at the top of SCAMMP, offers to the application layer an Application Programming Interface (API) to access these high-level context-aware information. This information is saved in a finite state machine and represents, in real time, the different context states (user, computational, physical). Hence, any application can easily integrate this information. The Acquisition layer is at the bottom of SCAMMP; its main role is to provide agents that encapsulate sensors.

Figure 1 also shows that mobile applications can share the same contextual information (decision logics) which in return shares data from the same sensors; this is shown by the different arrows pointing at different layers. It aims at reducing code redundancy hence decreasing the load on the system resources. Moreover, modules (state engines and agents) can be added/removed dynamically, hence guaranteeing the extensibility of the platform. Each layer offers services to the above layer via a set of commands (protocols). The different layers communicate as follows: the lower layer sends a notification (push mechanism) to the layer above, each time a sensor notification is received, thus indicating the presence of valid data. If the upper layer needs this data, it issues a request using a predefined protocol asking for the new updated data. At any time the upper layer is able to send a command to the lower layer requesting new data. The splitting of the system into different layers provides a high flexibility for layer hosting and thus the possibility to move the DL to the cloud.

2.1 DL

The main task of the DL is to provide a finite state machine that stores the different context (user, computational, physical environment) states in real time. It addresses the challenges of sharing high level context-aware information with the application layer while hiding the details of how they have been aggregated and augmented from different sensors. The core components of this layer are the state engines. Every time a new decision logic is needed (to maintain a new context), a new state engine is

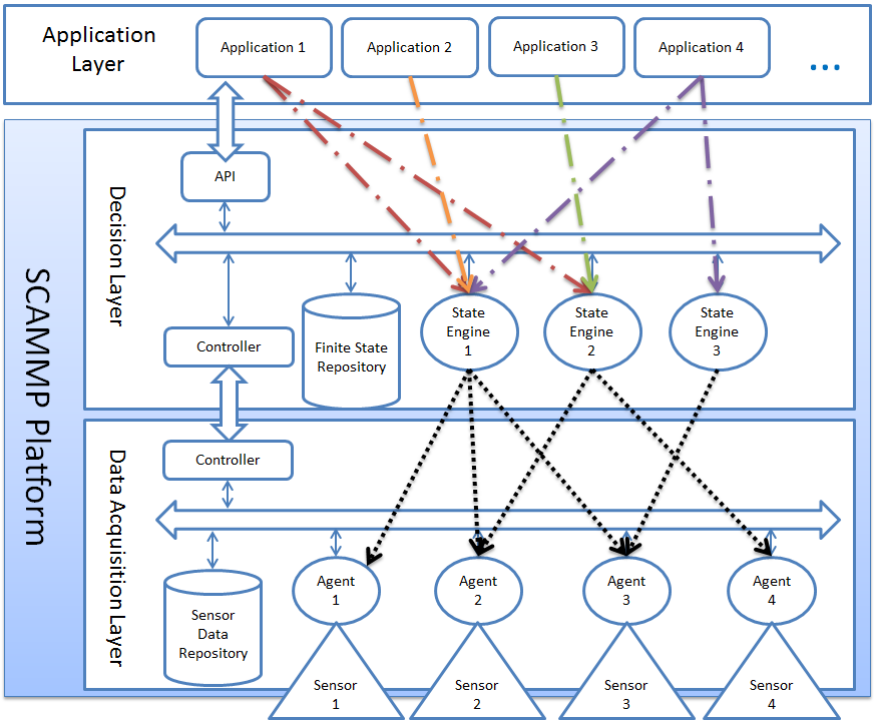


Figure 1: SCAMMP Detailed Architecture

injected to host this new decision logic. Figure 2 shows the components that constitute this layer. The API component represents the interface of the application layer and the controller component maintains an interface that can be used by DAL (lower layer). All remaining components are used exclusively within this layer. The API provides direct access to the files[2].Moreover, the DAL can be installed on handheld devices or on the cloud, it depends on the available bandwidth. If this layer is relocated to the cloud, then it can be shared (with little modification) with all application layers found on any mobile device.

2.1.1 State engine

Every user context state can have many attributes, for instance, *Home* is a user context state and *sleeping* is its attribute. Every state engine is built using a specific decision logic to generate a specific context state along with its attributes based on inputs from one or more sensor agents of the lower layer. The different finite state engine's outputs are stored in the finite state repository. When injecting a new finite state engine, it should be introduced to the controller in order to be registered before being operational. During the registration of a new state engine, a list of the connected sensor agents from the lower layer need to be specified.

Practically there is an abstract class called StateEngine; a new engine has to extend it and implement some of its methods by injecting the decision logic. It needs to register itself at the controller and specify the sensor agents it wants to listen to. This is all what needs to be done before the engine starts updating its finite state machine and producing high level contextual information. Each time a sensor agent of the acquisition layer sends a notification to announce the availability of new sensor data, the corresponding finite state engines will be notified via the controller. It is up to the state engines to decide whether to

[2]Again, in case the layer is relocated, other communication ways could be implemented

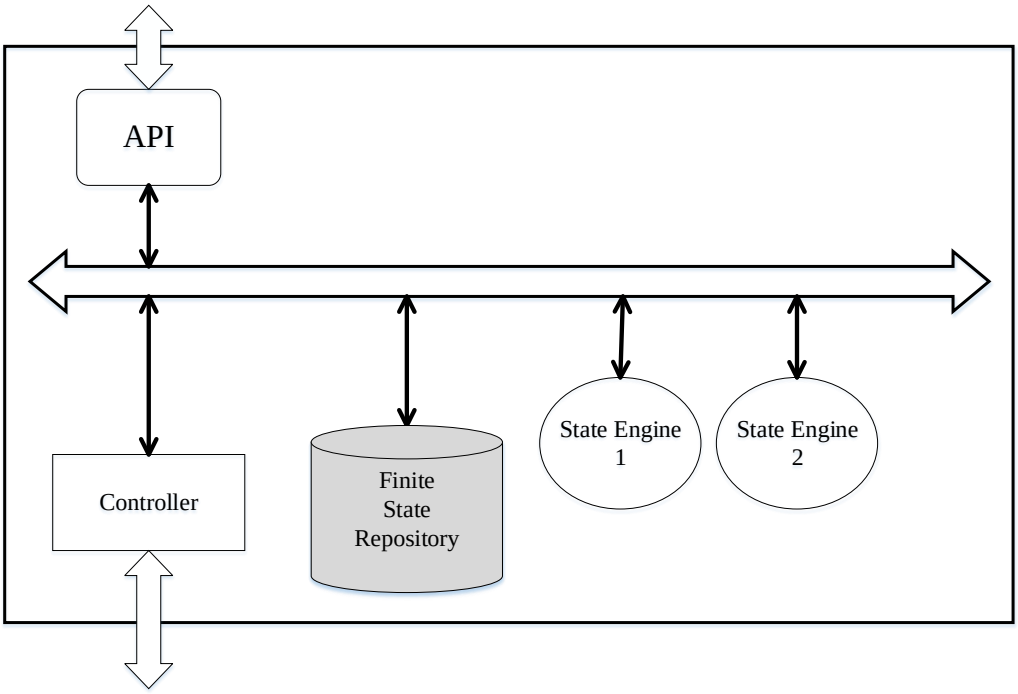
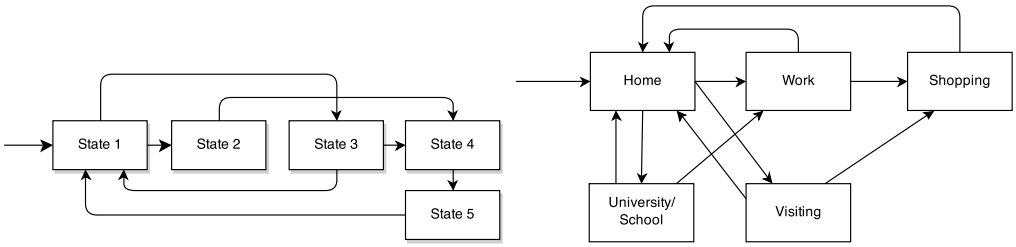


Figure 2: Architecture of the DL

request the data or not.

The kernel of the state engine is based on a finite state machine that updates the current context state. The state machine components (states and transitions) depend on the available state and its attributes; the state machine is updated and logical transitions are applied between states.



(a) State Machine Diagram (b) Location Finite State Machine

Figure 3: Finite State Machine

Figure 3a is a template state machine that is integrated within the state engine.In fact, all states are stored in the finite state repository of the DL (Figure 1) in XML format. For instance, we consider the location state machine with spatial user context states 3 : Home, University/School, Work, Visiting, and Shopping. The state machine of such states is almost common between people so that all states will return to Home state and most of them originate from Home State. The outcome of this engine is the current user state represented in an XML file that adapts the User State Schema and is stored in the state repository. The output becomes available to the application layer through the API component.

2.1.2 Application Programming Interface

The applications can access the high level context-aware information only through the API. The main role of the API is to provide the current values and the history of the different context states to the application layer. The API is implemented as a library (JAVA package) that has to be imported in any mobile application project that wants to access high level context-aware information provided by SCAMMP. Two different requests are provided by the API component, one request to list the available contexts and their attributes and the other one to request the current states /(values) of the different contexts or the history for a specific period of time.

2.1.3 Finite state repository

The Finite state repository is implemented as a storage system. It holds four different kinds of data, two of them are made available to the application layer via the API component and the other two are for internal use only. These four data types are implemented as XML entry lists that respectively:

1. contain an entry of every registered state engine.

2. store the current values and history of the different user states.

3. contain the output of every state engine.

4. declare for each agent (of the acquisition layer) the state engines that are attached to it.

As already mentioned, the last two data types are for internal use and are made available for the state engines. Since the history of the user state could become huge after a while, it can be archived and uploaded to the cloud (or any other remote storage), and still being available to the application layer. The following subsection (2.1.4) describes the XML schemas for the different data types.

2.1.4 XML Schemas

The XML files are saved in the repository and must follow a specific schema depending on the type of data stored as described above.
Figure 4 illustrates the state engine registration schema. Each state engine must have, in addition to its unique ID, a unique URI. The URI is used to reach the state engine data in case the DL is hosted in the cloud. The file also clearly declares the set of agents attached to the engine and its desired output.

The XML schema that represents the user state stores the current state or previous states so that the latest file is the current one and all others are user state history. The schema can be extended to include any kind of context information depending on the available state engines.

Figure 5a presents the unified schema used by state engines to format their output.

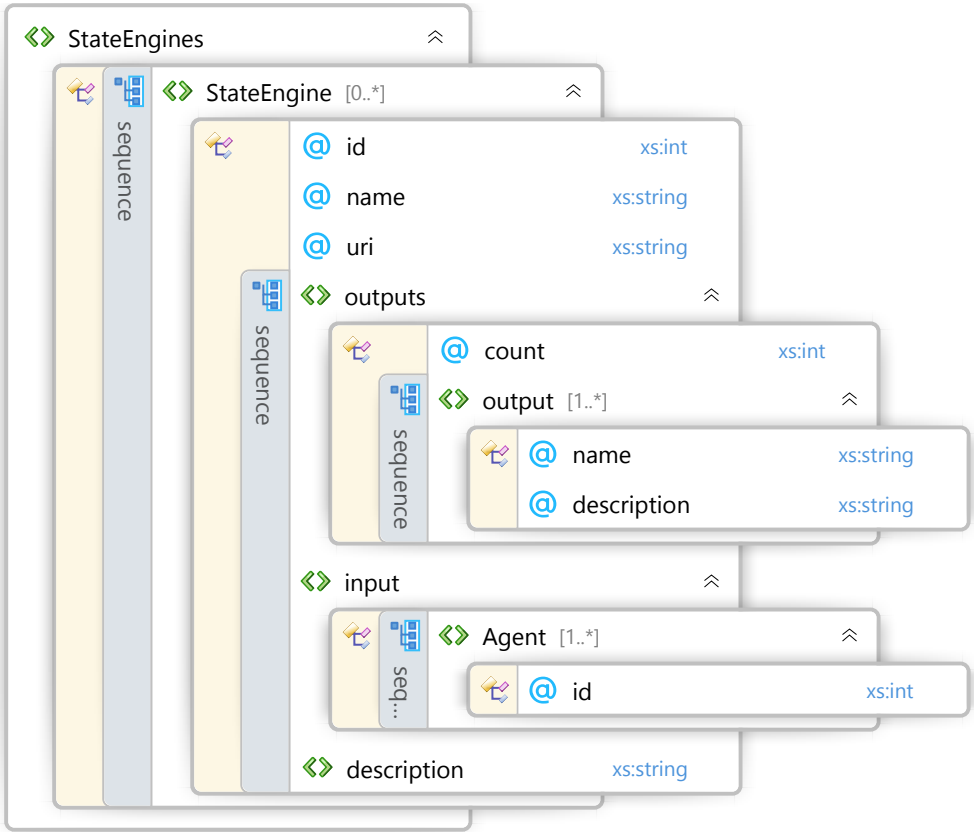
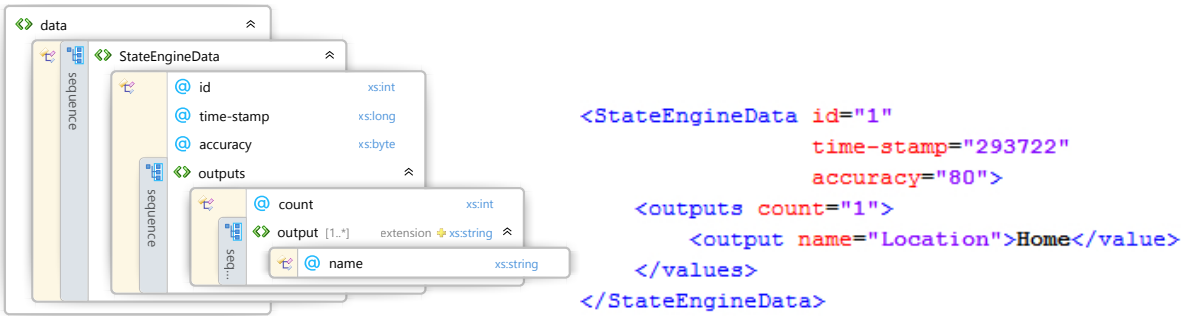


Figure 4: State Engine Registration XML Schema



(a) State Engine Data XML Schema (b) Sample output (Location Engine)

Figure 5: State Engine Data

The schema for Agent-Engine declares for each agent (of the acquisition layer) the state engines that are attached to it. This information is needed by the controller when receiving notifications from the lower layer to be forwarded to the engines concerned of the agent data update.

2.1.5 Naming Space

The repository stores its documents in a hierarchy of collections or folders. It contains three main directories to hold the four types of data (presented in section 2.1.3).

DecisionData Folder: contains two files: The user state file and the available states file. The user state

file stores the current state and the history of the user state following the 'Decision Data Schema'. The available state's files store all the values that the middleware can detect.

EngineData Folder: holds all the data produced by state engines. Each engine has a specific folder identified by its ID. The folder contains an XML file that follows the 'State Engine Data Schema'. Additional files required by the state engine might be added to this folder.

Registration Folder: This folder saves all available state engines in the 'registration.xml' file that adapts the 'State Engine Registration Schema'. It also contains the Agent-Engine attachment files, where each Agent has its own file that contains the state engines attached to it. These files follow the 'Agent-Engine Attachment Schema'.

Since the DL can be hosted in the cloud when needed, the repository files must have a unique URI so they can be reached remotely. Therefore, we suggest attaching a URI to each folder link in the repository: *www.SCAMMP.com/FiniteStateRepository/*.
For instance, data produced by Location engine can be found on the URI:
www.SCAMMP.com/FiniteStateRepository/EngineData/Engine1.

2.1.6 Controller

The controller maintains the communication with the lower layer and thus isolates both layers (DAL and DL) from each other. It receives notifications from the sensor agents layer (DAL) and forwards them to the corresponding state engines layer (DL). It forwards requests to DAL received from the available finite state engines(DL) and sends the answer back. The controller updates a list of active finite state engines and registers new injected state engines.

2.2 DAL

The main aim of DAL (Figure 6) is to provide the data collected from different sensors in a unified format. When injecting a new sensor whether physical or virtual, it will be encapsulated using a single dedicated agent. Once a sensor produces a new data, the corresponding agent will decide, based to a specific threshold whether to forward a notification to the layer above or not; if yes, the agent will read the data and save it in the data repository in a unified XML schema (Figure 7a). This data becomes available to the corresponding finite state engines of the upper layer through the controller.

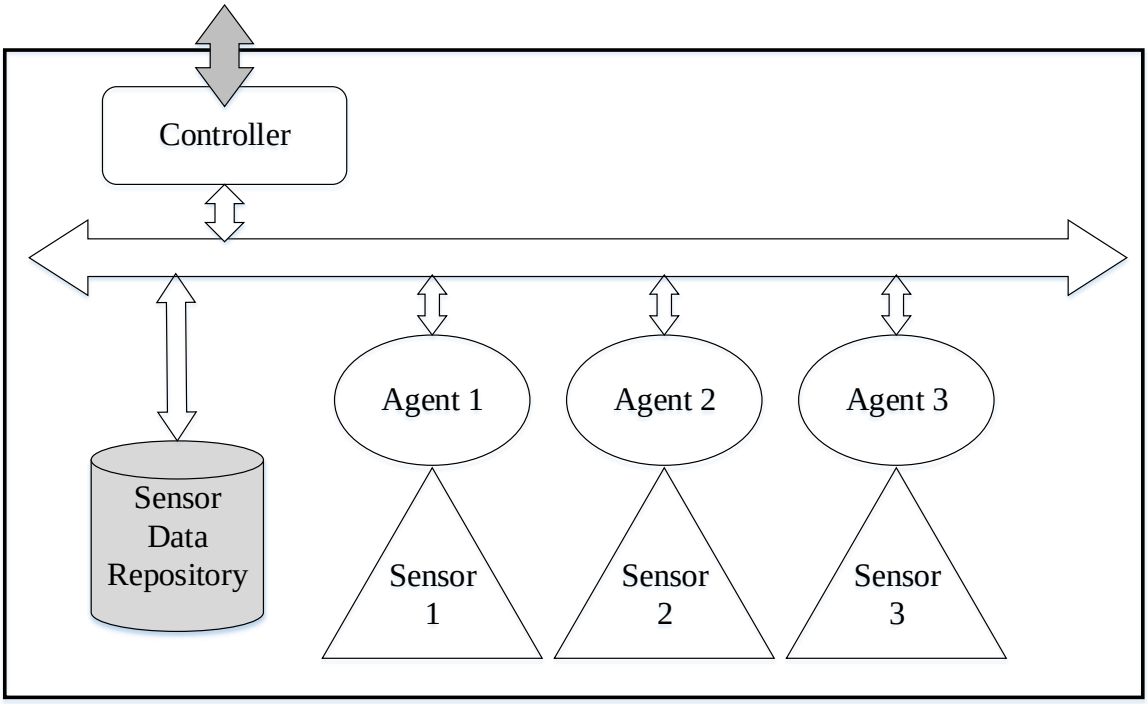


Figure 6: Architecture of the Acquisition layer

2.2.1 Agents

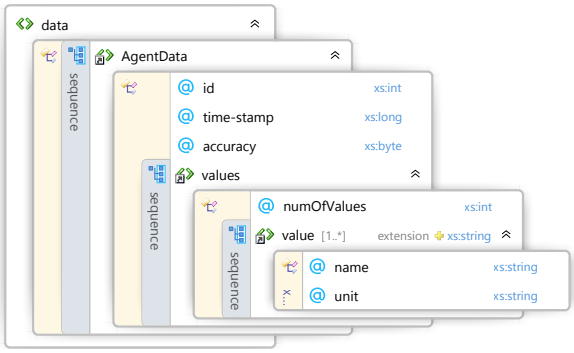
Most popular mobile operating systems provide sensor framework as an API. For instance powered mobile devices offers raw sensor data by using the Android sensor framework. This is part of the android hardware package and includes many classes and interfaces (SensorManager, Sensor, SensorEvent, SensorEventListener, etc.). The SCAMMP agents hide the OS API to offer a unified sensor data representation. A unified XML schema is made available to all agents to store the captured sensor data (Figure 7a)). Samples of real captured data are illustrated as three kinds of sensors (physical, virtual and logical) in figures 8a (Accelerometer sensor), 8b (Battery sensor) and 8c (Calendar sensor). Every sensor (whether physical, virtual or logical) is managed by a single agent. The agent implementation is sensor-dependent. Each time a new agent is injected, it has to be introduced to the controller.

2.2.2 Sensor Data Repository

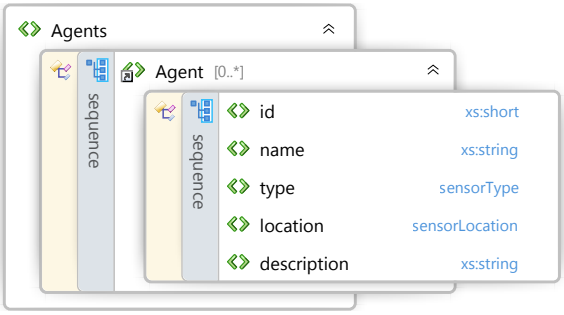
The Sensor Data Repository is an internal storage system that holds all information needed in the DAL. In order to organize the storage and for the retrieval of XML files, the sensor data repository is equipped with a repository manager to be used by the different agents. The repository contains three types of data: the agent registration file that contains all registered agents, the data produced by agents, and the configuration files (one for each agent).

2.2.3 XML Schemas

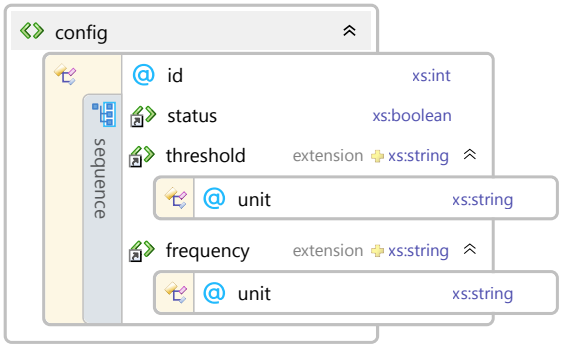
The XML files saved in the repository must follow a specific schema. Figure 7a presents the unified schema that is followed by agents to format sensor data. Figure 7b is the schema for agents registration file where 'sensorType' can have one of the values {physical, virtual, logical} and 'sensorLocation' can be one of {local, remote, cloud}. Finally, figure 7c is the schema for the agents configuration file.



(a) Agent Data



(b) Agent Directory



(c) Agent Configuration

Figure 7: Agent XML Schemas

```
<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>
```

(a) 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>
```

(b) Battery sensor

```
<?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>
```

(c) Calendar sensor

Figure 8: Agent captured data

2.2.4 Inter-process Communication

Communication between different components at the same layer as well as inter-process communication between the controllers at both layers is done through intent broadcasts and receivers. This requires the definition of several filters. All filter names are composed of three parts: The first part is a fixed prefix "SCAMMP". The second part is an abbreviation of the name of the sending layer (DL for Decision Layer and DAL for Data Acquisition-augmentaion layer). The third part specifies the sending component of a specific layer. Mainly components can send notifications, but the sender can be the DL controller because it might send messages to the state engines and to the controller at the lower layer (DAL). In this case, the third component prefix will be Controller_EngineX, where X identifies the state engine concerned with the message. In case the DL is relocated (e.g., hosted on the cloud), a new version of communication would be added using the same message format.

3 Case studies

The main goal of SCAMMP is to provide a middleware framework that offers high level context-aware information through a simple API to the application layer. But to achieve this goal, SCAMMP needs to be extended by injecting new decision logics and encapsulating new sensors. In this work, we present two case studies: the user location and the Mobile User Signature Extraction based on user behavioral Pattern [5] (MUSEP). In this section, we present how in general, SCAMMP can be extended, and then we present how each case study can be integrated within SCAMMP.

3.1 Extending SCAMMP

SCAMMP can be extended by injecting new decision logics and encapsulating new sensors. The rest of this section explains how to achieve this.

3.1.1 Encapsulating new sensors within the DAL

We need to encapsulate each sensor using one agent: this can be done by extending the abstract super class Agent: "public abstract class Agent extends Service "
Depending on the sensor type, there are two cases:

- Periodic Data Capture, which requires implementing the abstract method readSensorData()
- Event Based Capture, which requires the implementation a specific broadcast receiver

3.1.2 Injection of new state engines within the DL

The following steps need to be done in order to inject a new state engine:

- Extending the abstract super class StateEngine:"public abstract class StateEngine extends Service".
- Overriding the service methods of the class StateEngine to include the needed decision logic.
- Implementing a broadcast receiver that listens for SCAMMP DL Controller Engine.
- Registering the new state engine in the controller using the method:" RegisteredStateEngine registerEngine(String name, Output[] outputs, int[] input, String description)".

3.2 User Location case study

The aim of this case study is to determine the current location state (high level user context information) of the user, whether the user is at home, work or elsewhere. This can be done by injecting a new state agent at the DL called "Location State engine" and adding three agents.

3.2.1 Adding Agents

Three agents are needed to get the user's location. These three agents encapsulate the following sensors: Location (hardware/software sensor), network connection (software sensor) and calendar (logical sensor) sensors. These agents convert raw data generated by the encapsulated sensors into a unified XML data as depicted in figure 7a.

Location Agent: Most operating systems provide a location framework that calculates the location of the device. A software-based sensor found in most handheld devices that use the GPS, cell tower information, and the connected Wi-Fi network is responsible for calculating the user's location. It provides the following location data: longitude, latitude, altitude, accuracy in meters, and time. Figure 9 is an example of data generated 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 9: Location Agent Data

Network Connection Agent: This agent extracts the current user connected network type (e.g., Wi-Fi, Mobile Data network). It obtains this information from the "Connectivity Manager" integrated in the mobile OS. The main role of this agent is to send a notification whenever the user changes the network connection type. It returns the following data: connection type, connection SSID for Wi-Fi networks, and the connected cell towers. Figure 10 represents a real sample data for a Wi-Fi network connection with SSID 'Alfa' (a mobile operator in Lebanon).

```
<?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 10: Network Connection Agent Data

Calendar Agent: This agent encapsulates the logical sensor calendar. The information collected by this agent can be used (at the DL) to raise the certainty of the location obtained from other agents. Figure 11 is a sample data presenting a calendar event named 'Meet Manager'.

```
<?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 11: Calendar Agent Data

Each of the above agents is implemented by a class that extends the abstract class Agent; for instance the network agent is implemented as follows: "public class LocationAgent extends Agent " and overriding the method "public void onCreate() ".

3.2.2 Injection of Decision Logics

At the DL, a new state engine (called location state engine) needs to be injected, which hosts the decision logic. The agents presented above (location, network connection, Calendar) are attached to this state engine. The kernel (decision logic) of the location state engine can determine user's location (Home, Work, or elsewhere) using density-based clustering [6]. In fact, it goes through two phases: the learning phase and the active phase. During the learning phase, the engine remains for a period of time collecting the locations that the user frequently visits. In this phase, the "Location State Engine" uses data form the Location Agent only. After a period of two weeks of collecting data, the location history is clustered using density-based clustering into home, work, and elsewhere ("Elsewhere points" are the points recognized as noise by the clustering algorithm). Therefore, the clustering step is executed as a part of the learning phase. Thus, any location point can be classified in a certain location. After the clustering step is done, the network agent starts signaling the beginning of the active phase. Whenever a new connection is detected, this is mapped to one of the three high level locations (Home, Work, or Elsewhere) in an XML file following the 'Location-Connection Mapping Schema' (Figure 12).

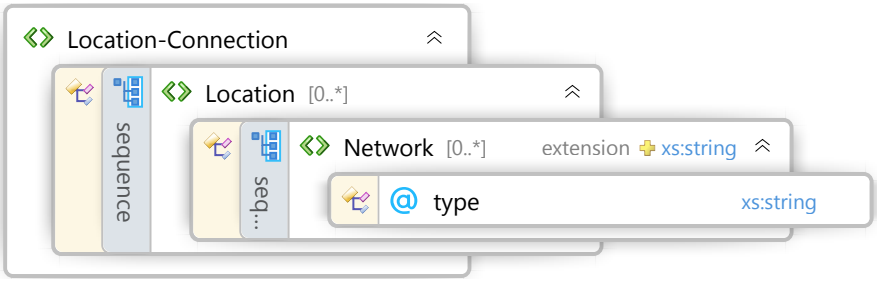


Figure 12: Location-Connection Mapping Schema

3.3 MUSEP

The MUSEP system aims at building an intrusion detection system based on user behavioral pattern. It is composed of three software components as depicted in figure 13. The three components are the learning component, the mathematical model and the intrusion detection component. The MUSEP is executed in two main phases: the learning phase and the intrusion detection phase. The rest of this section explains how MUSEP is implemented using SCAMMP.

3.3.1 Addition of Agents

Only one agent is needed. This agent encapsulates a software sensor namely, the User behavioral sensor. It is defined as a kind of user interaction with his/her phone. It has been decided to use the "WhatsApp" application as a proof of concept because of the frequent use of such application. Hence, each time the user uses the "WhatsApp" application, the agent records the start time and the duration in seconds.

3.3.2 Injection of Decision Logic

The intrusion detection logic is composed of two main components; the mathematical component and the intrusion detection component. In the mathematical component, data stored in the first phase will serve as a comparison tool for the decision phase. The input of this model is the start time of the activity (considered as abscissa x) collected at run time which is provided to the cubic spline function; the mathematical part of the overall algorithm. Next, a decision making tool will use both the data stored in the device, and the result of the cubic spline function in order to come up with the proper decision. The mathematical model component (the cubic spline function) is used in collaboration with the intrusion detection component to form the intrusion detection phase. In the later component, the algorithm will compare the ordinate "y" from the stored data with the result of the cubic spline. The stored data will also serve to determine a threshold by which the decision of the owner or adversary will be taken depending on the difference between the results.

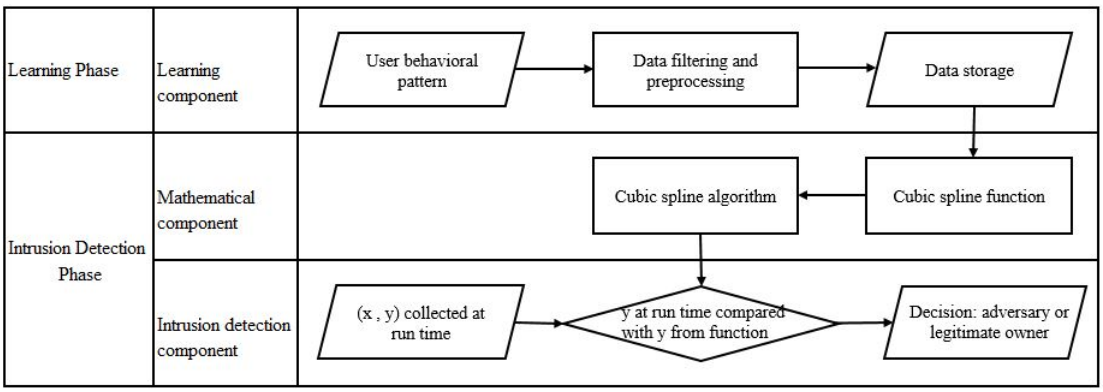


Figure 13: MUSEP Architecture

3.4 API

Therefore, any application that wants to integrate high level context-aware information about the current user location or the user authentication can do this by adding only three lines of codes. First the package (UserState) needs to be imported, and then a call to the following three static methods of the class API can be placed anywhere within the application:

- *API.getCurrentState()*: to get the user's last state
- *API.getBetween(GregorianCalendar from, GregorianCalendar to)*: to get user states between given dates
- *API.getAvailableStates()*: to get all available user states using the UserState format

4 Quantitative Performance Analysis

Handheld devices and despite the advances in battery technology, still have very limited energy resource and require frequent recharging. In addition, the processing power and the memory storage are considered as scarce resources. Hence, the installation of any middleware needs to be analyzed regarding the requirements put on of these three resources. This section presents a quantitative performance analysis of the resources (in terms of processing power, memory usage and power consumption) consumed by SCAMMP. SCAMMP is implemented as two separated applications one for each layer. Each application is implemented as a background multi-threaded service. We run the performance analysis by executing each case study a part and then running both of them together to test SCAMMP's scalability. We executed the running phase only, because it includes the learning phase.

4.1 Experimental Setup

One device was used. Below are the device specifications.

- **Device 1:** Samsung Ace Plus GT-S7500T, 1 GHZ processor android 2.3.6, 512MB RAM with SIM card (WiFi and 3G are used)

4.1.1 Tools and Methodology

The bash shell command "top" [7] is used to get CPU load (percentage) and the bash shell command "dumpsys meminfo" [8] is used to get the PSS (Proportional Share Size) value for SCAMMP which takes into consideration pages shared between processes, an android application has been developed to execute these commands periodically (every 3 seconds for CPU and 10 seconds for RAM). We used the Trepn Profiler [9] mobile application to gather the power consumption. The execution of these tests were done as follows:

- Run the location case study alone.
- Run MUSEP case study alone.
- Run both case studies together.

4.2 Running Phase

The running phase requires running both layers, the decision and the DAL. The DAL is responsible for the learning phase. Therefore we only evaluated the running phase because it covers both, the learning and the running phase. During this phase, the user changed his location and was connected to different networks that were new networks or recent ones to ensure executing all code paths of SCAMMP. The user also uses "WhatsApp" application during this phase in order to activate the user's behavioral agent. There are two active agents, the network and the user behavioral agent. We executed the data acquisition and DL for the following three different situations:

- Running the location case study, where the state engine "Location" and the network agent are active
- Running the MUSEP case study, where the state engine "MUSEP" and the agent user's behavioral agent are active
- Running both case studies, Location and MUSEP together, where both state engines and agents are active

4.2.1 CPU Load

We recorded the CPU load for both layers separately because they are implemented as two separated applications that reside in two distinct memory spaces.

Figure 14 depicts the percentage of CPU usage of the DAL for all three situations (Location, MUSEP and both) . It is clear that the CPU percentage usage is almost zero for long periods and at most (given by one record capture) it exceeds 14%. This is because the middleware (DAL only in this case) listens to events coming from sensor agents, so either the user is now attached to a new network or is using the "WhatsApp" application. The android API documents [10] states that the listeners are only considered active (in execution) when an event is received.
Figures 15 depicts the percentage of CPU usage of the DL for all three situations. It shows a similar result as the DAL with a maximum peak of 12% (less than the maximum of the DAL) this is due to the fact that the DL is not interested all the time in the data gathered by the DAL. In fact, if they do not exceed certain threshold, they will be ignored. Both figures 14 and 15 show that the injection of new states engines (by running both, the Location and MUSEP case studies) does not significantly increase the CPU load.

4.2.2 Memory Usage

Graphs in figures 16 and 17 , show that RAM usage recorded a maximum of 4912 KB for both case studies for all three situations (Location, MUSEP and both). The fluctuation of the memory usage is caused by the networks frequent connections and disconnections for the location case study and the use and release of the "WhatsApp" application. Each time the agents capture new data, JAVA objects are created (using the new operator) and later released by the garbage collector. This leads to frequent increasing and decreasing of allocated memory at the application heap. The same happens at the DL, whenever new data is received from the DAL, new objects are created and garbaged later. Figures 16 and 17 depict that when running two agents and two states engines, the memory usage will not remarkably increase. In fact, most of the claimed memory is due to the SCAMMP framework source code itself.

4.2.3 Power Consumption

The battery capacity is one of the most important components of any handheld device, but the battery life time depends on the application running. It is well known that I/O operations consume more power than CPU processing. In addition, handheld devices access mobile data network, hot spots and GPS. All of these consume a remarkable amount of battery energy. Many applications share the access to this device, that is why it's very difficult to attribute the amount of power consumed by a specific application. We used the mobile app "Trepen Profile"[9] to find out the amount of current (in mA) consumed by SCAMMP. Figure 18 depicts the current (in mA) consumed by SCAMMP while running both layers. The figure shows that the current consumed did not reach 0.1 mA, which means it is in the range of Micro-Ampere μA. In fact SCAMMP doesn't access GPS, Wi-Fi or mobile network data. It has a

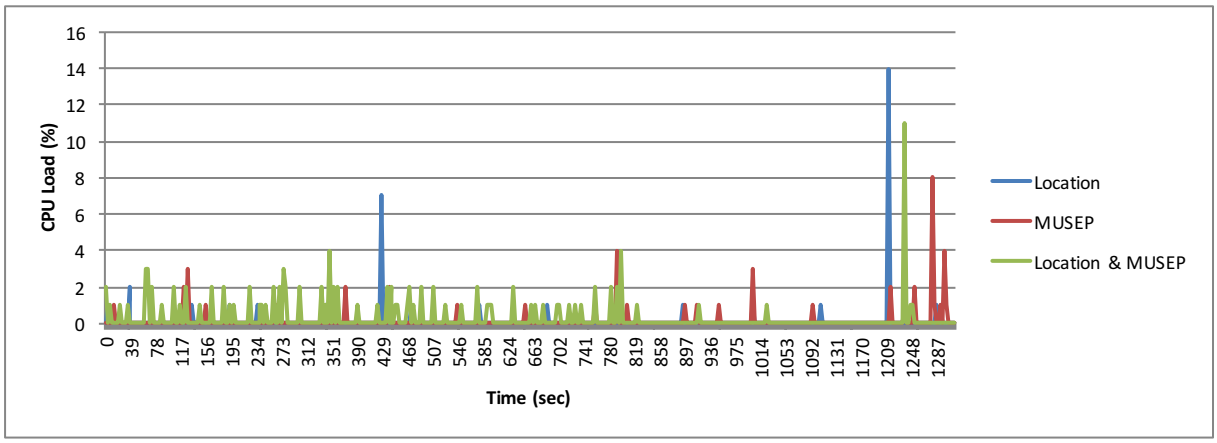


Figure 14: CPU-DAL Usage during the running phase

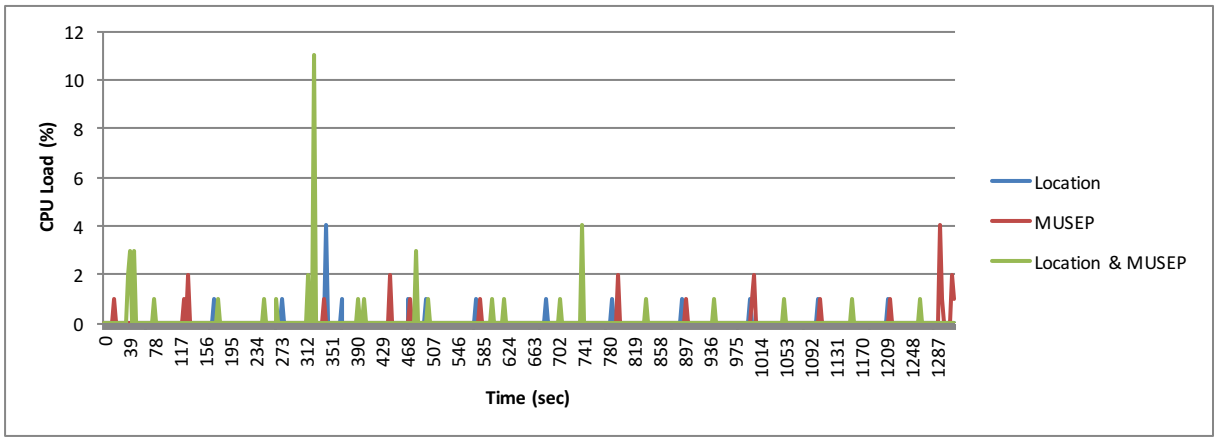


Figure 15: CPU-DL Usage during the running phase

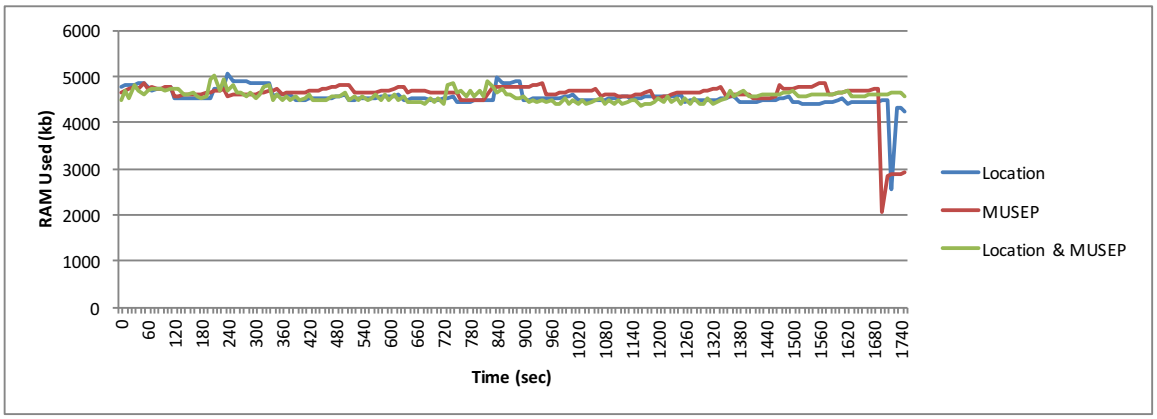


Figure 16: RAM-DAL Usage during the running phase

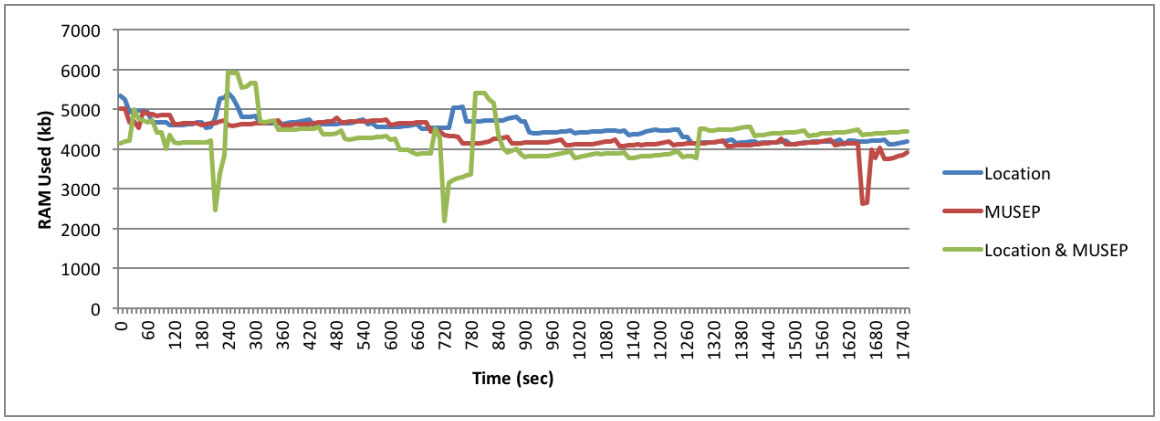


Figure 17: RAM-DL Usage during the running phase

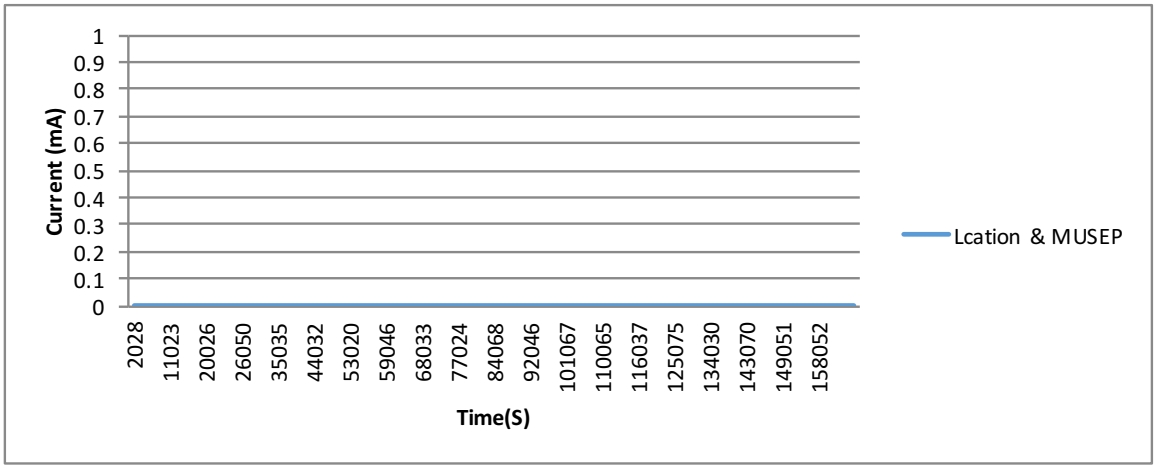


Figure 18: Current consumption in mA of SCAMMP (DAL+DL) during the running phase

listener that is activated whenever one of these devices sends a broadcast. This results in executing a method, which requires some CPU processing and memory access, thus it doesn't put a high load on the battery. Hence, we can conclude that SCAMMP does not put high additional overload on the CPU load, memory usage or battery life time. Even after injecting new state engines and the addition of new sensors, the additional overload remains acceptable.

5 Related Work

Developing mobile applications that are context-aware is a difficult task, this is due to the multiplicity and diversity of sensors found within mobile devices. Hence, many research works have proposed middlewares as a solutions to offer an abstraction layer between the operating system and the applications. In this chapter we present middleware solutions that provide context-aware information.
Baldauf et al.[11] presented a survey of context-aware systems and concluded that a common layered conceptual framework exists for most systems: Starting from the bottom (Sensors Layer) going through the Raw Data Retrieval Layer and Preprocessing Layer that increases the level of abstraction of contextual data. Furthermore, the Storage and Management Layer provide an interface to the Application Layer in order to obtain the needed data. While most systems are having common architecture, they differ in

the way they address the applications. Many proposed middle-ware architectures collect information for a wide range of context-aware applications such as smart homes, intelligent vehicles, and context aware hospitals. Other systems such as SCAMMP are targeting excessively handheld devices.
Henricksen et al. [12] presented the **PACE** middleware that offers heterogeneity, mobility, traceability and control, and deployment and configuration of new components. This middleware addresses many of the context-aware middleware requirements, except the scalability requirement which is not satisfied. This middleware is not targeting mobile devices exclusively, in fact, it offers context-aware information for general purpose systems. It is composed of three (3) layers: *Context Repositories Layer*, *Decision Support Tools Layer*, and *Application Components Layer*.
Dey et al. [13] introduced a framework that offers fast development of general context-aware applications. Using the *Context Widget*, *Interpreters*, *Aggregators*, and *Services*, it splits the context acquisition from the use of context within the application. The **Context Toolkit** aims at instantiating the framework, but its scalability, ease of deployment and configuration features are rather limited.
On the other hand, the generic context-aware framework **CMF** [14] is a scalable context-aware framework that allows processing and exchanging of heterogeneous context information. The *Context Source* applies reasoning techniques to aggregate data from different sensors and provides them to the *Context Provider*. This framework is based on user profiles information that is saved in the *User Management* component.
The **CORTEX** [15] project introduced the sentient object model that allows the development of context-aware applications in mobile ad hoc environments. A sentient object is defined as a mobile intelligent software component. It senses the surrounding environment through sensors and other sentient objects. It is composed of three parts: *Sensory Capture*, *Context Hierarchy*, and the *Inference Engine*. Thereafter, the proposed project was enhanced in [16] by injecting the reflection capability and the Service Discovery component. This has been tested by building an intelligent vehicle application.

SCAMMP somehow shares some similarities with the presented work above. It is similar in the way it collects data from various sensors, raises its abstraction level, and provides context information for the applications layer, but SCAMMP offers the following important additional features:

- Any application at the application layer has access to the different contexts provided by SCAMMP and this can be done using only three lines of code.
- SCAMMP can be easily extended and this again is considered at the design stage, by allowing the injection of new decision logics and the encapsulation of future sensors.
- The DL can be moved to the cloud and this is considered at the design stage. This allows the sharing of decision logics among different devices and not only among applications of the same device.
- A performance test a has been conducted using two case studies that showed the impact of SCAMMP on power, memory and CPU load remains acceptable when injecting new state engines and adding new agents.

6 Conclusion and future work

This paper presents a fully implemented middle-ware framework (on android platform). It includes the detailed architecture and design specifications as well as the communication protocol. In addition, ian implementation of two case studies is give to show how simple it is to extend SCAMMP and to evaluate the overhead in terms of power consumption, memory usage, and CPU load produced by SCAMMP.

The evaluation shows that SCAMMP's overhead due to power consumption is around 0%, processing power have peaks less than 14% at certain moments but is zero most of the time, and the memory usage did not exceed 5 MB. Moreover, SCAMMP offers high level context-aware information to the application layer through a well-defined API that is easily accessible by any application at the application layer through the addition of three lines of code. SCAMMP is by design constructed to allow the injection of new decision logics (that generate additional context-aware information) and the encapsulation of future sensors. All of them simplify the test of new decision logic ideas. That also allows the sharing of decision logics among researchers and engineers. In addition, it is relocatable allowing the DL to be hosted on the cloud. Finally, it uses standard protocol for inter-layer communication and URI for name spacing. All of the aspects of SCAMMP mentioned above distinguish it from the currently proposed middleware. In the future, we are intending to create a software community around SCAMMP that works on extending (injecting new decision logics and encapsulating new sensors) and sharing it. We are looking further into implementing it on an IOS platform.

References

[1] GSMA Intelligence, "Gsma mobile economy 2015 report," GSMA, Tech. Rep., 2015.

[2] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update, 2014-2019," Cisco White Paper, February 2015, http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf [Online; Accessed on September 10, 2016].

[3] H. S. Fatima Abdallah and A. Fadlallah, "Standardised scalable relocatable context-aware middleware for mobile applications (scammp)," in *Proc. of the 8th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM'14), Rome, Italy*, August 2014, pp. 56–61.

[4] "Scammp project website," http://www.scammp-project.info, July 2015, [Last accessed on 25/2/2016].

[5] B. El-Hage, "Mobile user signature extraction based on user behavioural pattern," Master's thesis, Arab Open University - Faculty of Computer Studies - Lebanon, 2015.

[6] H.-P. Kriegel, P. Kroger, J. Sander, and A. Zimek, "Density-based clustering," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 231–240, 2011. [Online]. Available: http://dx.doi.org/10.1002/widm.30

[7] "Top command manual page," Linux User's Manual, http://www.manpages.info/linux/top.1.html [Online; Accessed on September 10, 2016].

[8] Android developers, "Dumpsys system diagnostics," 2016, https://source.android.com/devices/tech/debug/dumpsys.html [Online; Accessed on September 10, 2016].

[9] "Trepn profiler," https://developer.qualcomm.com/software/trepn-power-profiler [Online; Accessed on September 10, 2016].

[10] "Android developers," 2016, http://developer.android.com/reference [Online; Accessed on September 10, 2016].

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

[12] K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam, "Middleware for distributed context-aware systems," in *Proc. of the OTM Confederated International Conferences on the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, Part I, Agia Napa, Cyprus*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, October-November 2005, vol. 3760, pp. 846–863.

[13] 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, pp. 97–166, 2001.

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

[15] G. Biegel and V. Cahill, "A framework for developing mobile, context-aware applications," in *Proc. of the 2nd IEEE Annual Conference on Pervasive Computing and Communications (PerCom'04), Orlendo, Florida, USA*. IEEE, March 2004, pp. 361–365.

[16] C.-F. Sørensen, M. Wu, T. Sivaharan, G. S. Blair, P. Okanda, A. Friday, and H. Duran-Limon, "A context-aware middleware for applications in mobile ad hoc environments," in *Proc. of the 2nd workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC'04), Toronto, Ontario, Canada*. ACM, October 2004, pp. 107–110.

Author Biography

Hassan M. Sbeity received the Dipl.-Ing degree in Electrical Engineering and Information Science form the Ruhr Universitt Bochum (Germany) in 1993. He worked many years in the development of PC I/O devices and their drivers at EBS in Germany. In 2001, he received the DEA (Masters) in Informatics, Modeling and Intensive Calculation from AUF and the Lebanese University in Beirut. In 2005, He received PhD in Informatics at "Valenciennes et du Hainaut Cambresis", LAMIH ROI (France) with collaboration of the University of Ghent (ELIS), Belgium. He is currently Assistant professor at the Arab Open University Lebanon branch (since 2003). His main research interests include open learning, computer architecture, application parallelization, Mobile computing, Multimedia applications, and memory optimization of embedded systems.

Mohammad G. Malli obtained the Engineering Diploma in Electrical and Electronics from the Lebanese University (Lebanon) in 2002. After that, he received the Master and PhD Degrees in Networking and Distributed Systems from the University of Nice Sophia Antipolis in 2003 and 2006 respectively. He is currently the coordinator of the ITC program in the Arab Open University – Lebanon branch. His research interests lie in the areas of Open Learning, Mobile Computing, Computer Networking, and Social Networking

Khalid S. Al-Tahat holds a B.Sc. in computer science from Yarmouk University in Jordan, an MSc. in AI from Malaysian University for Science and Technology and a PhD in Software Engineering from the National University of Malaysia. Currently, he is the coordinator of the information technology program at the Arab Open University- Jordan Branch. He worked at educational institutes in UK, Malaysia and Jordan. His research interest lies in the fields of modern software engineering, mobile computing, AI and teaching computing. He is an author of about 20 papers published in international journals, conference proceedings and invited book chapters.

Ahmad Fadlallah received the Engineering Diploma in Electrical Engineering and Electronics from the Lebanese University (Lebanon) in 2001. In 2003, he received the DEA (Masters) in Telecommunication Networks from AUF and the Lebanese University in Beirut. In 2008, he received a PhD in Computer science and Networking from Telecom ParisTech-France. He is currently Assistant-Professor at the Department of Information Technology and Computing at the Arab Open University – Lebanon branch (since 2008). His research interests lie in the areas of open learning, Mobile computing, mobile networks, multimedia services and computer & network security.

Mohamad H. Youssef received his BSc. in computer science from the Lebanese university, faculty of computer science in 2013. He obtained his MSc. studies in the information and decision support systems (IDSS) at the Lebanese university in 2015. He is currently in the process of starting his PhD.