A Service-Oriented Middleware for Context-Aware Applications

Luiz Olavo Bonino da Silva Santos
University of Twente
P.O. Box 217
7500AE – Enschede –
the Netherlands
+31 53 489 4454
l.o.bonino@ewi.utwente.nl

Remco Poortinga – van Wijnen
Telematica Instituut
P.O. Box 589
7500AN – Enschede –
the Netherlands
+31 53 485 0492
remco.poortinga@telin.nl

Peter Vink
Philips Research
P.O Box WB61
5656AA – Eindhoven –
the Netherlands
+31 40 274 9552
peter.vink@tass.nl

ABSTRACT
Context awareness has emerged as an important element in distributed computing. It offers mechanisms that allow applications to be aware of their environment and enable these applications to adjust their behavior to the current context. Considering the dynamic nature of context, the data flow of relevant contextual information can be significant. In order to keep track of this information flow, a flexible service mechanism should be available for the client applications. In this document we present a service-oriented middleware for context-aware applications. This middleware provides support to leverage the development of context-aware applications by providing a scripting-like approach for context-aware application development; allowing the subscription of rules containing context-based events and conditions and a notification to be sent when the specified context holds. Moreover, a domain-specific language has been developed to express these context-based rules.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed applications.*

General Terms
Design, Experimentation, Human Factors.

1. INTRODUCTION
Context awareness represents an important use of distributed computing and introduces a new class of smart applications. Awareness of a subject's surroundings and state helps applications to adapt their functionality depending on context changes and without (or with fewer) direct user interaction. However, the introduction of context-awareness in applications raises a series of new challenging requirements such as discovery and selection of context sources, interaction with these sources, and manipulation and interpretation of contextual information, amongst others. Coping with these challenges imposes a considerable effort for system designers and developers of ad-hoc context-aware solutions. To tackle these challenges, a flexible mechanism allowing user applications to easily specify the relevant changes in the environment is needed.

Commonly, context-aware systems involve the interaction of distributed, mobile, and heterogeneous applications and devices. In the approach presented here, we use the concepts and technologies of Service-Oriented Computing to cope with the issues of distribution, mobility, and heterogeneity. Here, we present a middleware that provides: *(i)* facilities to manage contextual information and context sources and *(ii)* support to client applications to subscribe context-based rules and receive notifications when a specified context holds.

The remainder of this paper is structured as follows. Section 2 introduces the service-oriented middleware for context-aware applications. Section 3 details the Context Management Service while section 4 details the Awareness and Notification Service. Section 5 presents a use case scenario supported by the current implementation of the middleware. And section 6 concludes and positions our approach to related work found in literature.

2. THE CONTEXT-AWARE MIDDLEWARE
Our context-aware middleware is the integration of two components, the Context Management Service (CMS) [9] and the Awareness and Notification Service (ANS) [3]. The Context Management Service supports context sources to publish their contextual information to be used by context-aware applications and services. The Awareness and Notification Service offers a rule-based facility allowing client applications to subscribe rules containing context-based conditions and receive notification when the specified context holds. shows the architecture of the proposed combined middleware regarding the connections with client applications and context sources.

CMS supports context sources to publish their contextual information to be used by context-aware applications; it provides the infrastructure and data model for context sources to publish their contextual information. Applications or services interested in particular types of context information, such as user location, device's status, amongst others, also use the CMS to find the appropriate context source for providing this information. The application can then query the context source for information or subscribe to it, in which case the application will receive a notification whenever there is a change in the context information it is interested in.

MPAC 2007, November 26–30, 2007, Newport Beach, CA, USA.

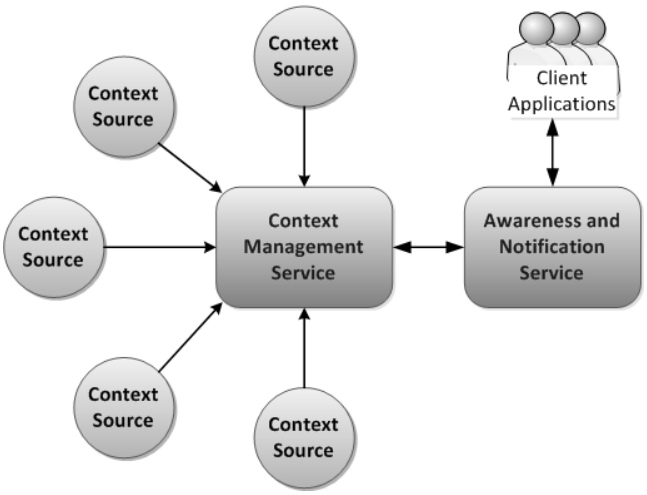

Figure 1 - Context-aware middleware architecture.

The client applications shown in can subscribe monitoring rules to ANS. ANS then determines the types of context information it needs for each given rule and tries to find, and subscribe to the context source(s) that can provide the needed context information. Based on the information from the context source ANS then, continuously determines whether the condition specified in the monitoring rule holds. If the condition holds, ANS proceeds to the notification phase according to the notification information specified in the rule, i.e., the message and the message's target.

An example of such a rule specifies that when John enters his home, his mother Maria should be notified. In this simple example, ANS finds the appropriate context source by asking CMS for a reference to a context source that can provide location information. ANS then monitors the location by subscribing to that context source for the location information of John. The context source informs ANS whenever the location of John changes. When John enters his home, ANS sends the specified notification to Maria.

Our approach considers that changes in the application's environment are modeled by means of Event-Condition-Action (ECA) rules [1][2]. Our domain-specific language has been developed to define context and context events supporting the specification of context-based reactive behaviors.

The architectural design of both CMS and ANS follows the Service-Oriented Architecture (SOA) principles; each component is implemented as a web service relying on standards such as SOAP, WSDL, web service dynamic discovery (WS-Discovery), WS-Eventing, amongst others.

The following sections provide a more detailed description of both CMS and ANS.

3. THE CONTEXT MANAGEMENT SERVICE

The Context Management Service (CMS) [9] provides the necessary middleware to manage context sources and their contextual information. CMS consists of a set of well-defined component interfaces and data model for representing context. Figure 2 shows the main components defined in the CMS architecture as well as the various operations that can be invoked on these components. In the figure the methods that can be invoked are shown as text on the arrows between the different components, where the arrows indicate the direction of the typical interactions that take place between the components.

The CMS architecture consists of three main component types; context sources (CS), context broker (CB), and context consumers (CC); each having their own responsibilities:

- CB: keeps track of all the context sources within a (network) domain and acts as a service directory for context sources.
- CS: provides context information of a specific type to context consumers. When a CS starts up, it registers with the CB at startup, describing the type of context it produces.
- CC: uses the context information provided by the context source. Finds the appropriate context sources by asking the context broker.

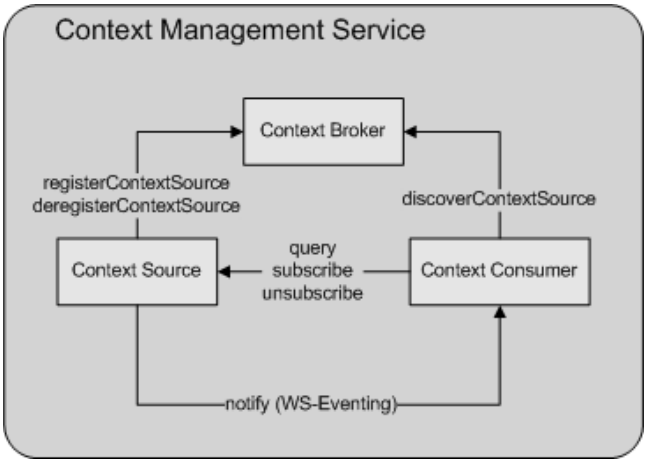

Figure 2: Context Management Service (CMS) architecture.

A context source provides two main modes of operation:

- Synchronous (request/response): a context consumer asks for context matching certain specified criteria, the context source responds to that query by providing the appropriate context information.
- Asynchronous (publish/subscribe): a context consumer indicates to the context source that it wants to be informed of the changes in context information meeting certain specified criteria, the context source informs the context consumer of these changes whenever they occur.

The former mode of operation allows a context consumer to ask context sources for relevant context information when that is needed by the context consumer, while the latter allows a context consumer to act reactively to changes in context information since it will be notified of the changes in the context information it is interested in.

Next to a standardized interface, context sources in CMS also use a standardized context representation format.

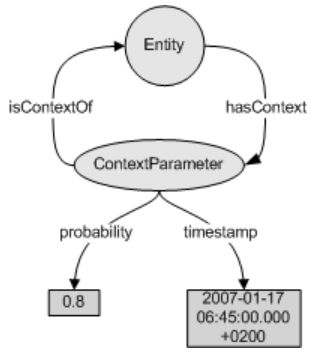

Figure 3: Generic ContextParameter.

CMS uses ontologies (OWL/RDF) for context representation. Figure 3 shows the generic ContextParameter concept from which all other types of context information are derived. A ContextParameter has one or more object type properties referring to a (subclass of an) Entity. Every ContextParameter also has metadata associated with it, in the form of data or object type properties, which tell something about that particular ContextParameter, such as the probability of correctness or its timestamp.

For specific types of context information, subclasses of the ContextParameter concept are derived. The isContextOf property is subclassed as well for specific context information.

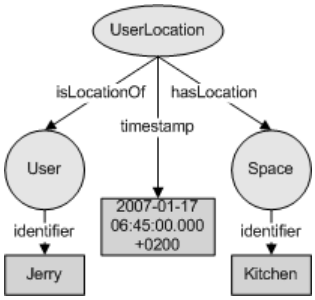

Figure 4: Example of context information.

Figure 4 shows an example for representing the location of a user. Here the subclass of ContextParameter is called UserLocation, which has two subproperties of isContextOf defined: an *isLocationOf* object property that refers to the User for which the location is specified, and a *hasLocation* object property that refers to the Space that is the location of the user. The *timestamp* indicates when this context information was determined. The specific example in the figure tells us that in the early morning of January 17th 2007, Jerry was in the Kitchen. The probability and other attributes are omitted in this example. Note that if no intermediate *UserLocation* (or *ContextParameter*) concept was used to link a User with a Space; it would be impossible to reliably represent when Jerry was in the kitchen. This problem is depicted in Figure 5. Linking a timestamp property to the relation is not possible here; neither would linking to the user or the space provide the same information, since the precise meaning of that timestamp property would then be ambiguous.

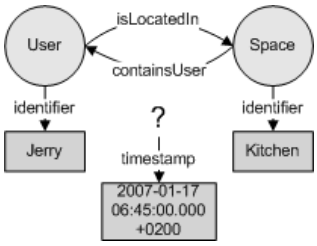

Figure 5: Metadata problem if not using ContextParameter.

A context consumer uses SPARQL [10] for querying a context source, allowing a context consumer to ask only for the specific context information it is interested in. Taking the context from Figure 4 as a further example: the following SPARQL query shows how to ask for all the last known locations of users:

```
PREFIX amigo: <http://amigo.../Amigo.owl#>
PREFIX context: <http://amigo.../ContextTransport.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?username ?roomname WHERE {
  ?user rdf:type amigo:User .
  ?user context:identifier ?username .
  ?userloc rdf:type context:UserLocation .
  ?userloc context:isLocationOf ?user .
  ?userloc context:isLocatedIn ?room .
  ?room context:identifier ?roomname
}
```

Any component implementing the context source interface, registering itself with the broker, and providing context information in the appropriate format is considered a context source. This implies that components can be a context source and context consumer at the same time, such as a component that combines different types of (typically lower-level) context information from other context sources into new (higher-level) context information for republishing. An example of this is a Location Management Service (LMS), which keeps track of users by combining information from different technologies, such as RFID, Acoustic positioning, Bluetooth, etc. This LMS would be a context consumer for the context sources providing the lower-level context for the individual technologies, but would be a context source for providing the combined measurements as User locations.

The project in which the Context Management Service is developed is targeted at the future connected home; assuming that all future Consumer Electronics devices will have network (IP) connectivity. To accommodate different specifications from the manufacturers of these devices, two different platforms were used for implementing the CMS: OSGi/Java [4][5] and .NET. By providing CMS implementations for different platforms, manufacturers are free to choose the implementation that is best suited for their devices. The context broker was implemented in OSGi only. For processing queries from context consumers, .NET context sources typically use SemWeb [16], while OSGi context sources use Jena [17] for the same purpose.

4. THE AWARENESS AND NOTIFICATIONS SERVICE

The Awareness and Notification Service (ANS) supports developers in adding context-awareness capabilities to their applications. Using ANS, developers do not have to deal with monitoring, controlling and managing contextual information inside their applications. This avoids the necessity of creating specific context-awareness features for each application and, therefore fosters rapid development of context-aware applications. Applications are only responsible for registering so-called *monitoring rules*. These rules specify the context to be monitored and the notification to be sent once the expected context holds.

Once the client application has subscribed and started the monitoring rule, ANS starts gathering the required contextual information. In the case that the triggering condition contained in the monitoring rule holds, ANS proceeds to notify the client application according to the notification message specified in the rule. An example of such rule specifies that John should be notified when his children arrive at home. In this example, the ANS monitors the location of John's children and notifies him when they enter the home.

Following the Event-Control-Action pattern described in [1], four main sub-components are present in ANS as depicted in . *EventMonitor* receives context data events from context sources through the CMS. *EventMonitor* sends these events to the *Controller* that monitors them and evaluates the registered rules. When the triggering condition of the rule is evaluated to true, the *Notifier* is called to perform the suitable action. The action (in this case, the notification) also depends on the user's context, for instance, if the user is in a meeting the notification can be sent via SMS to his mobile phone instead of via e-mail; which would be the case if he were in his office. The subscribed rules and the ontologies used in ANS are stored in a *KnowledgeRepository* and made available for both *RuleManager* and *EventMonitor*.

As previously mentioned the architectural design of ANS follows the Service-Oriented Architecture (SOA) principles and is implemented as a web service. The external entities with which ANS interacts are also implemented as web services, such as the client applications and the CMS. In the internal perspective, the ANS implementation follows the OSGi component based framework approach [4]. The current implementation of ANS uses the Oscar OSGi Framework [5].

ANS exports two interfaces available both in the Oscar framework and as web service's interfaces through WSDL:

• *IManageRule*, used by client applications to manage rules, and;

• *IReceiveContext*, which is a call back mechanism to receive information from context sources through the CMS. In addition, ANS reacts to Web Service events generated by CMS.

The RuleManager component is externally accessed via the *IManageRule* interface. The *RuleManager* provides facilities for (un)subscribing, updating, starting and stopping rules. When a client application wants to register a rule, it sends the rule to the *RuleManager* that is responsible for parsing, validating and storing the incoming rule. In the parsing and validating phases, the *RuleManager* translates the given user rules to reaction rules that can be handled by the *Controller*.

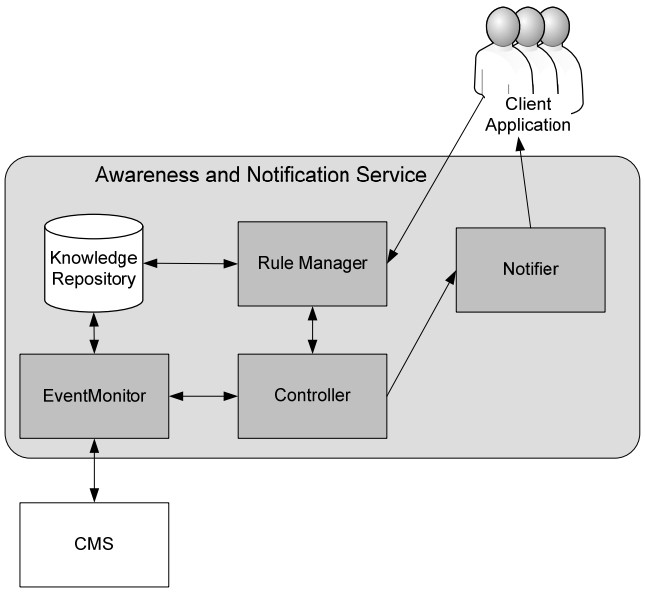

Figure 6 - The ANS architecture

The rules received by the *RuleManager* from client applications are expressed in the ANS domain-specific ECA language called ECA-DL [6]. *RuleManager* transforms this ECA-DL rule into a rule that can be handled by the underlying rule-engine. Currently, ANS uses the JESS rule engine [11].

Once a rule is registered, it is available to ANS but not yet subject to monitoring, i.e., the rule is only registered in the system's *Knowledge Repository* but its triggering condition is not susceptible to be evaluated. An application has to "start" the rule to start the evaluation of the rule's triggering condition. When a registered rule is started the *RuleManager* sends it to the *Controller*. The *Controller* then extracts the context variables (the eventing part) and other context variables of the rule and submits these events to the *EventMonitor*. Figure 7 shows a fragment of an UML Sequence Diagram depicting the message exchange of the rule subscription.

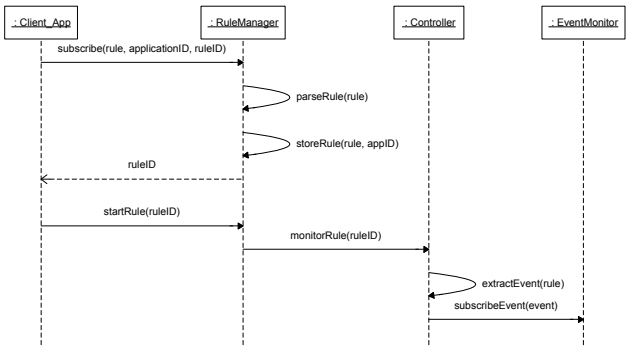

Figure 7 – UML fragment of the rule subscription activity

The main functionality of the *EventMonitor* is to provide easy access to context data. The *EventMonitor* provides to other ANS components a mechanism for subscribing to or querying for context data. As an example, if the *Controller* needs to monitor the battery level of a device, the *EventMonitor*, through CMS,

searches for an appropriate context source that could supply this information. Once found, *EventMonitor* subscribes to the request battery level data and informs the *Controller* of events containing the request data.

The *EventMonitor* maintains a subscription to the corresponding context source for every event that the *Controller* has requested. The maintenance of a list relating events and context sources is important to avoid redundant subscriptions. To accomplish this, the *EventMonitor* analyzes the requested subscriptions searching for overlaps in the subscription's requirements. An example of a requirement overlap is when different rules request the same event from a single context source. In this case the *EventMonitor* keeps only one subscription to the context source.

After subscribing the events contained in the rule to the *EventMonitor*, the *Controller* starts receiving notifications of the occurrence of these events. For every event notification received, the *Controller* evaluates the new information against the notification rules. When the rule's condition is evaluated true, the *Controller* invokes the *Notifier*.

The task of the *Notifier* is to send notifications to applications and users. For each notification it determines the appropriate level of intensity before sending the notification. Users create an individual user notification profile using the *User Modeling and Profiling Service* (UMPS) (not shown here due to space restrictions). The profile describes how and when a user wants to be notified. Before sending the notification, ANS checks the notification profile of the user that is to be notified. Based on this profile, ANS sends a notification with the intensity information and the receiving application notifies the user accordingly. If a user has not created a notification profile, a standard profile is assigned. An example is that a user had set in his notification profile an intensity of 1 (ambient notification) when he is at home. In this case, when he receives a notification at home, the receiving application notifies him by changing the ambient lights.

The notification of a user works as follows. When an active rule evaluates to true, the *Controller* sends a notification event to the *Notifier*. The *Notifier* transforms the event into a notification. An event encompasses the message, the UserIDs of the users that are to be notified and references to applications. First the *Notifier* determines the right intensity for the notification. It does so by retrieving the relevant user notification profiles. In case additional context parameters are necessary in order to determine the intensity of the notification, the *Notifier* queries the *EventMonitor*. This is for example the case if a user has specified not to be notified at home. In this case the *Notifier* queries the *EventMonitor* for the current location of the user. Once the notification profiles are evaluated and the intensity of the notification is determined, the *Notifier* sends the notification to the application. It is then the task of the application that receives the notification to interpret the level of intensity, e.g. to change the color of an ambient light in order to send a notification with a low level of intensity to a user.

The ANS' domain-specific rule language, namely ECA-DL, allows application developers to conveniently enhance their applications with reactive context aware behavior by using a scripting format. This relieves the developer from writing programming code inside his application to deal with context management; this is handled by the ANS. ECA-DL has been developed following the event-condition-action (ECA) pattern. Rules in ECA-DL are composed of an *Event* part that models an occurrence of interest in the context, a *Condition* part that specifies a condition that must hold prior the execution of the action, and an *Action* part which consists of reactive invocations.

ECA-DL is defined upon two complementary information and behavior foundations. Information foundation refers to the representation of the applications' universe of discourse, i.e., a domain ontology. For example, we should be able to express within ECA rules whether people are in the house or not, whether objects are plugged in or not, whether persons and objects are collocated, among others. Behavior foundation of the ECA language refers to the dynamics of rule execution, i.e., how and when a rule should be executed and what are the elements of the language that should be used to perform a particular piece of reactive behavior.

For the information part, a domain ontology should be referenced. In the scope of the Amigo project [12], where this work is being developed, the Amigo Ontology is used. ANS assumes that one is only allowed to use a piece of knowledge in the ECA rule, if this has been previously defined in the ontology. If the ontology does not define the concept co-location, for example, this concept cannot be referenced in ECA rules (due to space limitations, more details about ECA-DL can be found in [6]).

A simple, non-parameterized rule is composed by the basic structure:

```
Upon <event-expression>
When <condition-expression>
Do <action>
<lifetime>
```

5. USE CASE

In order to illustrate and evaluate the usability of our middleware, we present a use case in the scope of Amigo project [12]. The Amigo project aims at developing open, standardized, interoperable middleware and intelligent user services for the networked home environment. In our use case example, the following scenario is envisioned:

> "*During the week days, when John leaves the house without his laptop on a day with a scheduled meeting, he should be informed.*"

In this use case location sensors are used as context sources and can provide the location of people and devices. Moreover, John's personal calendar application is also used as a context source registered in CMS to provide information about his meetings. The rule to be subscribed to ANS is:

```
Upon EnterFalse(isAtHome(User.John))
When    isAtHome(Device.Laptop123)    and
   isOwnedBy(Device.Laptop123,
   User.John) and hasMeeting(User.John)
Do  notify(User.John,  "You  forgot  your
   laptop at home")
Lifetime from "Monday" to "Friday"
```

After the subscription and the parsing phases being carried out by the *RuleManager* and the *Controller* components, ANS queries CMS for suitable context sources. CMS searches among its database of registered context source for a match. Once found, CMS returns the reference of the context source to ANS which subscribes to the necessary contextual information.

For the *isAtHome* event part of the example, ANS asks CMS for a context source that keeps track of which users are at home. When CMS returns the reference for a context source, ANS subscribes to it with a query parameterized for John. After the ANS' subscription to the contextual information, the context source informs ANS every time the result of the query changes. This allows ANS to test if the answer changed from TRUE to FALSE (the *EnterFalse* part of the Upon clause). In this example, the transition *EnterFalse* is used to express when John is no longer at home. ANS performs similar subscriptions to other contextual information for the conditions in the *When* clause.

All the information from incoming events is evaluated by the *Controller* by pushing it into the internal rule engine of ANS [11] for evaluation. Since the triggering element is the event, i.e., the element in the *Upon* clause, the conditions in the *When* clause are only evaluated if the event occurs. Once all the events and conditions specified in the ECA rule are met, John will be notified with the message '*You forgot your laptop at home*'. The notification is sent to John according to his notification preferences and current context. For example, depending on the notification intensity he defined, the notification could be delivered by SMS or by email.

6. CONCLUSION

Current approaches for context-aware support middleware [13][14][15] provide ways to subscribe to and manage context data, but fall short on providing a decision support; i.e., providing a mechanism for applications to specify what context data they are interested in and what to do in case a given situation occurs. Moreover, these approaches do not offer a reaction process based on the users' context. Gaia [7] presents a distributed middleware infrastructure using the abstraction of a meta operating system, but although context is an important concept in Gaia, it is externalized and is not a class entity that drives the behavior of the system.

In this paper we presented a middleware platform that supports the development of context-aware applications. The context middleware is composed of two main components, namely CMS and ANS. The Context Management Service (CMS) component is responsible for handling context sources' registration and management of contextual information. The Awareness and Notification Service (ANS) handles the subscriptions of notification rules from client applications. These rules contain events and conditions based on contextual information that ANS gathers from context sources through CMS. CMS and ANS share an ontology for context representation allowing the expression of relations between entities and data.

Context is used in this middleware not only as event and conditional elements of the rules but also to evaluate how the client will be notified based on the notification parameters of user's preferences. Both ANS and CMS have been designed following the SOA guidelines and are implemented as web services using communication protocols such as SOAP, WSDL, WS-Discovery, and WS-Eventing, among others.

The middleware presented here by the integration of ANS and CMS is being evaluated in the scope of the Amigo project. The evaluation performed by implementing several use case scenarios and verifying their effectiveness in the home labs available for the project. A final result of this evaluation is expected by the end of the project, in February 2008.

7. ACKNOWLEDGMENTS

The work reported here is supported by the European Commission as part of the IST-IP Amigo project under contract IST-004182.

8. REFERENCES

[1] Dockhorn Costa, P., Pires, L. F., Sinderen, M., *Architectural Patterns for Context-Aware Services Platforms* in Proceedings of the 2nd International Workshop on Ubiquitous Computing (IWUC 2005), Miami, May 2005, pp 3-19.

[2] Ipina, D., Katsiri, E., *An ECA Rule-Matching Service for Simpler Development of Reactive Applications*. Published as a supplement to the Proc. of Middleware 2001 at IEEE Distributed Systems Online, Vol. 2, No. 7, November 2001.

[3] Bonino da Silva Santos, L.O., Ramparany, F., Dockhorn Costa, P., Vink, P., Etter, R., Broens, T., *A Service Architecture for Context Awareness and Reaction Provisioning*, 2nd Modeling, Design, and Analysis for Service-Oriented Architecture Workshop (MDA4SOA 2007), Salt Lake City, USA, July 13th 2007.

[4] OSGi Consortium, http://www.osgi.org.

[5] Oscar OSGi Framework - http://forge.objectweb.org/projects/oscar/

[6] Dockhorn Costa, P., Ferreira Pires, L., van Sinderen, M., Broens, T., Controlling Services in a Mobile Context-Aware Infrastructure, in Proceedings of the 2nd Workshop on Context Awareness for Proactive Systems – CAPS 2006, Kassel, Germany, June 2006.

[7] Román, M., et al, A Middleware Infrastructure for Active Spaces. IEEE Pervasive Computing, 1(4):74-82, Oct-Dec 2002.

[8] Chan, A.T.S., Chuang, S.N., Mobi-PADS: A Reflective Middleware for Context-Aware Mobile Computing, IEEE Transactions on Software Engineering, vol. 29, n. 12, December 2003.

[9] Ramparany, F., Poortinga, R., Stikic, M., Schmalenströer, J., Prante, T., An open Context Management Infrastructure, 3rd IET International Conference on Intelligent Environments 2007 – IE07, Ulm, Germany, September 24-25 2007.

[10] http://www.w3.org/TR/rdf-sparql-query/

[11] JESS – the Rule Engine for the Java Platform. Available at http://herzberg.ca.sandia.gov/jess/.

[12] Ambient Intelligence for the Networked Home Environment – Amigo, http://www.hitech-projects.com/euprojects/amigo/

[13] Bardram, J. E., "Applications of Context-Aware Computing in Hospital Work – Examples and Design Principles" in Proceedings of the ACM Symposium on Applied Computing, 2004, pp. 1574-1579.

[14] Chen, H., "An Intelligent Broker Architecture for Context-Aware Systems", PhD proposal in Computer Science, University of Maryland, Baltimore, USA, 2003.

[15] Dey, A. K., "Providing Architectural Support for Building Context-Aware Applications", PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[16] http://razor.occams.info/code/semweb/

[17] http://jena.sourceforge.net/