A Service Architecture for Context Awareness and Reaction Provisioning

Luiz Olavo Bonino da Silva Santos[1], Fano Ramparany[2], Patricia Dockhorn Costa[3], Peter Vink[4], Richard Etter[5], Tom Broens[6]
University of Twente[1,3,6], France Telecom[2], Philips Research[4], Fraunhofer IPSI[5]
{l.o.bonino1[1],p.dockhorncosta[3],broens[6]}@ewi.utwente.nl, fano.ramparany@orange-ftgroup.com[2], peter.vink@philips.com[4], etter@ipsi.fraunhofer.de[5]

Abstract

Context awareness has emerged as an important element in distributed computing. It offers mechanisms allowing applications to be aware of their environment and enabling them to adjust their behavior to the current context. In order to keep track of the relevant context information, a flexible service mechanism should be available for the client applications. In this paper we present a service architecture to provide context-awareness capabilities to users and client applications. Moreover, the service is able to react depending on the user's preferences and context. The conditions for the reaction and the reaction itself are defined in rules the users submit to the service by means of a convenient rule language.

1. Introduction

Context awareness represents an important use of distributed computing and introduces a new class of smart applications. Awareness of user's surroundings and state helps applications to adapt their functionality depending on context changes and without direct user interaction. However, introducing context-awareness in applications demands a series of features such as: discovery and selection of context sources and interaction with them; manipulation and interpretation of contextual information; among others. These factors make difficult for system designers and developers to introduce ad-hoc context-awareness solutions. To tackle these requirements, a flexible mechanism allowing user applications to easily specify the relevant changes in the environment is of need.

Commonly, context-aware systems involve the interaction of distributed, mobile and heterogeneous applications and devices. Therefore, the use of concepts and technologies of Service-Oriented Computing can support tackling these issues of distribution, mobility and heterogeneity.

In this paper we present an Awareness and Reactive Service – ARS – following a rule-based approach which provides reactions (notifications, service/application invocations) depending on users' context. The following section (2) presents the context-aware reactive service. Section 3 details the Awareness and Reactive Service's architecture. Section 4 introduces the ARS rule language. Section 5 exemplifies the creation of a client application. Section 6 presents a use case in a home to home scenario and section 7 gives conclusion and points to future work.

2. The Awareness and Reactive Service

The Awareness and Reactive Service supports developers in adding context-awareness capabilities to their applications. Thus 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 fostering a rapid development. Applications are only responsible of registering monitoring rules. These rules specify which context should be monitored and which reaction should be triggered once the expected context holds.

Once the client application has subscribed the monitoring rule, ARS starts gathering the required contextual information. In the case that the triggering condition contained in the monitoring rule holds, ARS proceeds to the reactive phase according to the reaction specified in the rule. An example of such rule specifies that when the user John enters his home, the lights should be turned on. In this example, the ARS monitors the location of John and when he enters his home, the service invokes the illumination control system to turn on the lights.

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.

3. The ARS architecture

Following the Event-Control-Action pattern described in [1], three main parts are present in ARS as depicted in Figure 1. The EventMonitor receives context data events from context sources through the Context Management

2007 IEEE Congress on Services (SERVICES 2007)
0-7695-2926-7/07 $25.00 © 2007 IEEE

Service (CMS). The EventMonitor sends these events to the Manager that monitors them and evaluates the registered rules. If the triggering condition of the rule is evaluated true, the Reactor is triggered to perform the suitable action. The subscribed rules and the ontologies used in ARS are stores in the KnowledgeRepository and made available for both the Rule Manager and the EventMonitor.

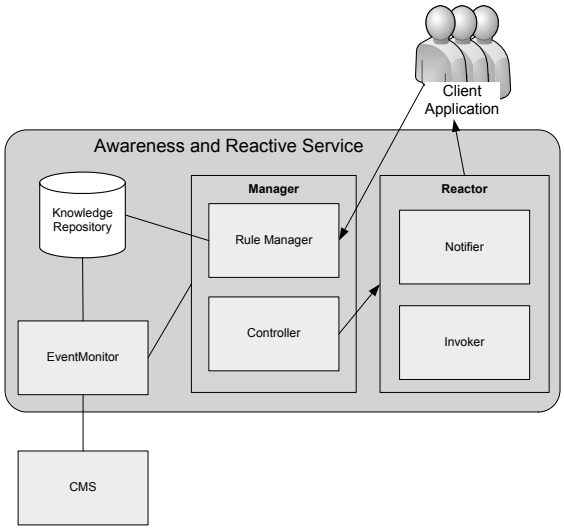

Figure 1 –The ARS architecture

The architectural design of ARS follows the Service-Oriented Architecture (SOA) principles. The service is implemented as a web service relying on standards such as SOAP, WSDL, UDDI and XML. The external entities with which ARS interacts are also implemented as web services, such as the client applications and the CMS. In the internal perspective, the ARS implementation follows the OSGi component based framework approach [3]. The current implementation of ARS uses the Oscar OSGi Framework [15].

The ARS web service exports two interfaces:

- IManageRule, used by client applications to manage rules, and;
- IReceiveContext, which is a call back mechanism to receive information from context sources through the context management service.

These interfaces are available both in the Oscar framework and as web service's interfaces through WSDL.

The Manager is the central component of ARS and is responsible to handle the client's rule subscriptions. The Manager is composed of two sub-components, namely the Controller and the RuleManager. The RuleManager is externally accessed via the IManageRule interface. The RuleManager provides facilities for unsubscribing, updating, starting and stoping 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 entered user rules to reaction rules that can be handled by the Controller.

The rules received by the RuleManager from client applications are expressed in the ARS domain-specific ECA language called ECA-DL [11]. The RuleManager transforms this ECA-DL rule into a rule that can be handled by the underlying rule-engine. Currently, ARS uses the JESS rule engine [10].

Once a rule is registered, it is available to the ARS but not yet subjected of monitoring, i.e., the rule is only registered in the system but its triggering condition is not going to be evaluated. To start evaluating the rule's triggering condition is necessary to "start" the rule. When a registered rule is started the RuleManager sends it to the Controller. The Controller then extracts the eventing part of the rule and subscribes these events to the EventMonitor to search for and request information from context sources. Figure 2 shows a fragment of an UML Sequence Diagram depicting the message exchange for subscribing a rule.

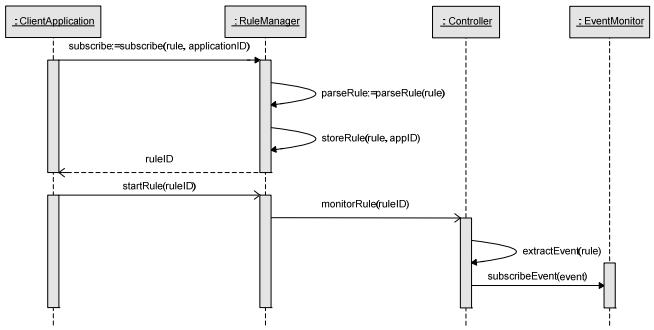

Figure 2 – UML fragment of the rule subscription activity

The main functionality of the EventMonitor is to provide easy access to context data. This includes searching for context sources, selecting a context source, registering to the context source eventing mechanism and deregistering when a context source is no longer needed.

The EventMonitor provides to other ARS 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 a single context source can provide two different events. 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 notification rules. When the rule's condition is evaluated true, the Controller invokes the Reactor requesting the appropriate reaction.

In the current version of ARS, the Reactor is composed of two sub-elements, namely the Notifier and the Invoker. Depending on the type of reaction requested in the user rule, the Notifier or the Invoker or both are activated.

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. The intensity of notifications is based on the current context of users and their personal notification preferences.

If users want to receive personalized notifications, they create individual user notification profiles. A user notification profile defines which level of intensity is appropriate in which context. The intensity of notifications can be based on availability of the user that is to be notified, or where the user is currently located. A further option is to take the co-presence of other persons into account. In general, the notification profiles are dynamic and allow users to take into account all available context data. When editing a personal user notification profile, a user is free to combine the different parameters in order to specify the appropriate level of intensity for certain situations. In case a user defines settings that are conflicting, an implemented conflict strategy implemented in ARS resolves the issue. A user can use one of the predefined profiles or refine one the profiles. If a user has not created a notification profile, a standard profile is used.

The notification of a user works as follows. If a rule in the Controller 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 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 notification to a user.

The Invoker is responsible for reacting to the occurrence of situations by invoking services and applications. It calls the service's method passing the specified parameters. The introduction of the Invoker relieves the applications to take actions based on the event's notifications.

4. The rule language

The ARS rule language, coined 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 notifications. This is handled by the ARS service when initiating the rules.

ECA-DL is a domain specific language developed with the purpose of specifying event-condition-action (ECA) rules to be used in context-aware scenarios. Rules in ECA-DL are composed by 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 foundations: 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 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 [9], where this work is being developed, the Amigo ontology is referenced. ARS 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.

When designing the ARS ECA rule language, high attention has been paid to the following qualities:

- Expressive power: the language permits the specification of complex event relations. It allows the use of relational operator predicates (e.g., < , >, =), and the use of logical

connectives (e.g., AND, OR, NOT) on conditions to build compound conditions.
- Convenient use for application developers: It provides high-level constructs that facilitate event compositions.
- Extensibility: The language allows the addition of new predicates to accommodate events being defined on demand.

4.1. Basic Concepts

Context changes are described as changes in situation states. Situations represent specific instances of context information, typically high level context information. Situations may be defined upon other Situations or Facts [12].

Facts define current "state of affairs" in the user's environment. Example of a Fact is Jerry is married to Maria. The situation context abstraction allows application developers and users to leverage on the fact abstraction in order to derive high-level context information. Example of a situation is *isOccupied*, derived from the fact "Jerry is cooking" or "Maria is working". Situations may be built upon other situations, for example, *isAvailable* may be defined as not *isOccupied* and *isReachable*. Facts and situations are defined as part of the overall information models (ontologies).

An event expresses a change in state, which is of interest to particular applications or users. For example, an application may be interested to know when Jerry enters the TV room, when Jerry and Roberto get close, or when Pablo becomes online in the instant message system. These scenarios refer to changes of state:

- Jerry enters TV room: State (Jerry is not in TV room) followed by State (Jerry is TV room);
- Jerry and Roberto get close to each other: State (Jerry and Roberto are far from each other) followed by State (Jerry and Roberto are close to each other);
- Pablo becomes online: State (Pablo is offline) followed by State (Pablo is online).

In ECA-DL, we allow the definition of events in two ways: expressed as an explicit change of state, or expressed as an event description. Consider the examples defined above. We may express the situation "Jerry enters room" as *EnterTrue (LocatedIn (Jerry, TVRoom))* or *EnterRoom (Jerry, TVRoom)*. Similarly, we may define "Jerry and Roberto get close" as *EnterTrue (CloseBy (Jerry, Roberto))* or *GetClose (Jerry, Roberto)*. Finally, we may define "Pablo becomes online" as *EnterTrue (OnLine(Pablo))* or *BecomesOnline(Pablo)*.

When defining in the ECA rule the change of state, for example using the state transitions *EnterTrue* and *EnterFalse*, the application developer expresses events by explicitly defining the state transition for a given situation. Conversely, application developers may express events by using pre-defined event descriptions. This requires these events to be previously defined in the ontology.

In ECA-DL there are three possible states (true, false and unknown) and six state transitions. The unknown state accommodates uncertainty of context information (when the value of context information in unknown). Figure 3 presents the possible state transitions.

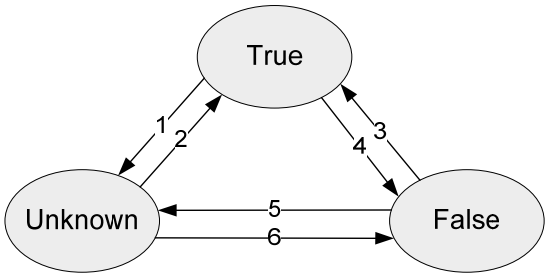

Figure 3 – State transitions for a situation

Events can be any of the following transitions, for a given situation S:

- EnterTrue(S) – transition 2 or 3
- EnterFalse(S) – transition 4 or 6
- EnterUnknown(S) – transition 1 or 5
- ExitTrue(S) – transition 1 or 4
- ExitFalse(S) – transition 3 or 5
- ExitUnknown(S) – transition 2 or 6
- TrueToFalse(S) – transition 4
- TrueToUnknown(S) - transition 1
- FalseToTrue(S) – transition 3
- FalseToUnknown(S) – transition 5
- UnknownToTrue(S) – transition 2
- UnknownToFalse(S) – transition 6
- Changed (S) – any transition

The condition part of a rule describes extra conditions that must hold prior the invocation of a notification. It differs from the event part, since it does not define a change of state it only specifies additional requirements (states) that must hold. In addition to specifying an event, application developers may be interested in more specific situations. For example, an application may need to be notified upon Jerry entering the TV room, under the condition that Jerry should be alone. The condition that Jerry should be alone does not define a state transition; it just expresses extra requirements for the notification to be invoked.

Conditions are logical expressions that inquire whether (a combination of) situations are true or false. Situations refer to concepts defined in the ontology.

Actions describe operations that should be invoked when both the event and conditions parts of rules are fulfilled. In ARS, actions are currently restricted to notification and invocation operations. The way

notifications are delivered depends on the users' preferences and context.

4.2. Syntax and Semantics

The condition part of ECA rules comprises two parts: an event part that defines a relevant situation change; and a precondition part that defines a logical expression that must hold following the event and prior to the execution of the notification. Both events and pre-conditions are defined in terms of situation and facts.

Each rule is associated with a lifetime, which can be always, once, from <start> to <end>, to <end>, <n> times, frequency <n> times per <period>. Always defines that a rule should be triggered whenever events and conditions turn true. Once defines that a rule should be triggered one time, and then should be deactivated. From <start> to <end> defines a period for the rule to be active for triggering. To <end> defines when a rule should be deactivated. <n> times says the number of times a rule should be triggers. Frequency <n> times per <period> defines the number of times a rule should be triggered in a certain period of time, for example, once a day or twice a week.

Events are defined in the Upon clause, while conditions are specified in the When clause and finally, notifications are specified in the Do clause. In the case there are no conditions to be specified the When clause may be omitted.

ECA rules can be parameterized. Parameterization is necessary when the rule should be applied to a collection of entities. It would be cumbersome to write a rule for each target entity. For example, a medical clinic would like to apply a general rule (notify when sugar levels go above 110) to all patients suffering from diabetes. Parameterization allows the specification of a single rule to be executed for all the diabetic patients. We have introduced the Scope clause to define rule parameterization.

From the necessity of filtering entities' collections respecting a certain condition, we have defined the Select clause. It allows the selection of a subset of a collection respecting context and/or attributes constraints. For example, it may be necessary to select all users that are in the house and taking a shower, or we would like to select all devices that are currently being used, or even all the patients of the clinics that have diabetes. Figure 4 depicts the ECA language metamodel in UML.

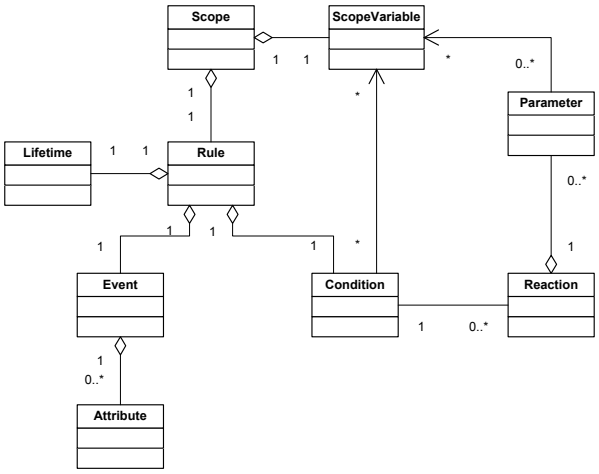

Figure 4 – The ECA-DL metamodel

A simple, non-parameterized rule is composed by the basic structure:
Upon <event-expression>
When <condition-expression>
Do <action>
<lifetime>

We have defined an XML schema representing the ECA-DL syntax. This schema allows the validation of ECA-DL rules written as XML documents. An example is the following rule: "Invoke the lightning service to turn on the bedroom lights when baby Anna cries and notifies Luciana".

In ECA-DL:

Upon EnterTrue(isCrying(Anna)
Do Notify (Luciana, "Anna is crying") AND Invoke (LightingService, TurnOnLights(BedroomAnna))
Always

In XML, this rule would look like:

```
<ECARule
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\ECAXML.xsd">
<upon>
    <event name="isCrying" state_transition="EnterTrue">
        <param>
            <literal value="Anna"/>
        </param>
    </event>
</upon>
<do>
    <reaction type="Notify">
        <param>
            <literal value="Luciana"/>
        </param>
        <param>
        <literal value="Anna is crying"/>
        </param>
    </reaction>
```

```
<reaction type="Invoke">
    <param>
        <service name="LightingService"/>
    </param>
    <param>
        <method name="TurnOnLights">
            <param>
                <literal value=”BedroomAnna”/>
            </param>
        </method>
    </param>
</reaction>
</do>
<lifetime value="always"/>
</ECARule>
```

5. Creating an ARS client

The ARS offers an API to be used by client applications' developers. A developer willing to create a client application should invoke the IManageRule interface. This interface is connected to the aforementioned RuleManager and defines the following functionality:

- Subscribing, updating and unsubscribing rules;
- Querying of registered rules;
- Enabling and disabling rules.

The code snippet below presents the IManageRule interface.

```
public interface IManageRule {
    public int subscribe (Rule rule, String applicationID)
                                throws RuleFormatException;
    public boolean unsubscribe (int ruleID);
    public boolean updateRule (int ruleID, Rule rule);

    public Rule queryRule (int ruleID);

    public boolean startRule (int ruleID);
    public boolean stopRule (int ruleID);
}
```

The first three methods allow the subscription, update and deletion of rules. The subscribe method requests a rule expressed in the ECA-DL XML format and a unique identification of the subscribing application. After parsing and validating the rule, the method returns a unique identification of the rule. The rule identification is used in the other methods as the reference for the subscribed rule. If, during the validation of the rule ARS finds a malformed element, a RuleFormatException error is thrown causing the subscription to fail.

The queryRule method is used to retrieve the ECA-DL XML of an already subscribed rule using its unique identification.

The last two methods are used to start and stop rules. Starting a rule means that it is enabled and the evaluation engine checks whether its defined conditions has been reached. Stopping a rule only disables it causing the evaluating engine to ignore that rule. To remove a rule from the system, the unsubscribe method should be used.

In the case of a rule requesting a notification as the reaction, the client application should implement also the interface INotifyApplication as presented next.

```
public interface INotifyApplication {
    public void notify (String message, String userID, int
    intensity);
}
```

The notify method is called by the ARS when a notification is triggered. The message and userID parameters are defined in the rule. The intensity parameter is based on the current context of the user and his personal notification preference.

Similarly to the IManageRule interface, the INotifyApplication should be implemented and publish by the client application as a web service.

6. A home to home use case

In this section we briefly introduce a use case that is currently being implemented in the framework of the Amigo project [13], as an assessment of a number of open source components that will be made available by the project to support development of ambient intelligent applications.

The overall goal of the concept is to provide persons with a tangible interface to stay aware of activities and presence of their contacts and to initiate ambient or explicit communication between the user and their contacts. One tangible interface that is being investigated is the Awareness Globe[1] which early prototype is displayed in Figure 5.

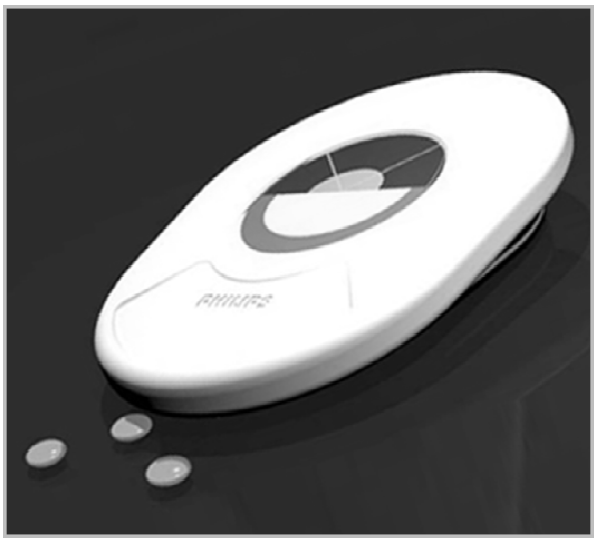

Figure 5 – The Awareness Globe prototype

[1] Awareness Globe – Philips Design, The Netherlands

Figure 6 depicts the Awareness Globe interface and Figure 7 shows how the device presents the information by color change.

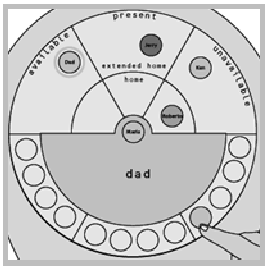

Figure 6 – The Awareness Globe interface

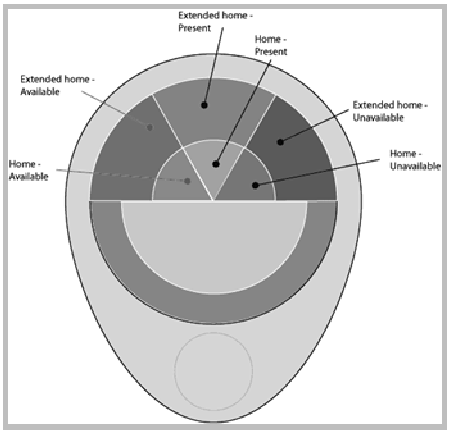

Figure 7 – The Awareness Globe information presentation

As illustration of the use of the Awareness Globe we describe the following scenario: "Maria comes back home. By placing her car key on the Awareness Globe, she accesses the in-house services and applications and can control her own availability and presence to interact with the outside world."

A home to home communication system integrating the ARS is currently being developed in the framework of the Amigo project. The overall architecture of this system is introduced in Figure 8.

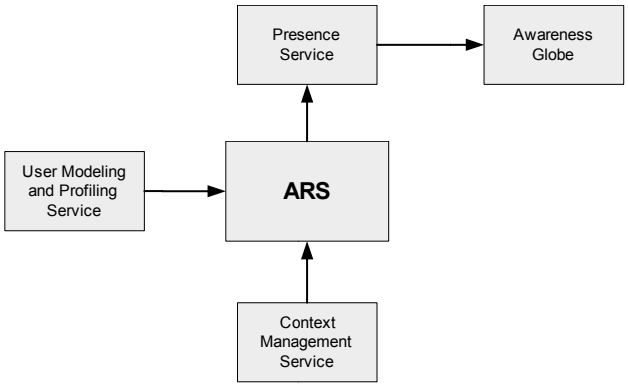

Figure 8 – The home to home communication system

The ARS interacts with the Presence Service which role is to control the Awareness Globe functions. As depicted in Figure 7, the Awareness Globe presents the ARS' notifications in a graphical form allowing an easy and fast visualization of a user contact's locations.

One main advantage of the ARS service is the separation between the way the notification and invocation will be supported in the physical environment and the awareness and reactive service itself that the ARS provides. For example, with almost the same ARS rule (by only changing the destination of the reaction) it is possible to replace the Awareness Globe with the Ambilight [14] (peripheral multi-hued ambience lighting) of a TV.

7. Conclusions and future work

In this paper, an awareness and reactive service has been presented. The service enables developers to rapidly implement applications that allow users to be aware of their environment. Applications register monitoring rules (i.e. specification of conditions based on context and corresponding actions) that specify what their users are interested in, by using a convenient rule language. Hereby, the management or monitoring of context data is delegated to ARS. The entered rules are continuously evaluated by ARS and proper reaction is produced when one of the rules evaluate to true. Additionally, the service provides notifications that are tailored to the user's situation (context). Before the notification service sends a notification, the appropriate intensity for the notification is determined. This ensures that the notification service provides notifications that are as unobtrusive for users as possible and as intrusive as necessary.

At the light of the first experience the Amigo project has gained in using the ARS, the main advantages of its use include:

- Its flexibility to handle the notification, presence and availability in the action part of ARS rules.
- Its capability to set rules and to notify about the presence of other users.
- Its capability to subscribe to events related to changes in other users context.
- Its capability to take into account users context, profile and preferences for notifying, in the condition part of ARS rules.

A first version of ARS has been implemented using Java and the generic rule engine JESS. The implementation is based on the Oscar OSGi component based framework to get a clear separation of concerns between the ARS sub-components. The current implementation version, called ANS, can be accessed at [16]. Remotely ARS can be accessed using standard web service technologies.

Current approaches for context-aware support middleware [17][18][19] provide ways to subscribe to and manage context data. But they fall short on providing a decision support, i.e., providing a mechanism that applications could specify what context data they are interested in and what to do in the case a given situation is achieved. Moreover, these approaches do not offer a reaction process based of the users' context.

The ARS rule language currently does not provide means to specify temporal ordering of events. The language will be extended to support temporal aspects, such as sequencing and concurrency of events. Secondly context models will be considered. Moreover, the current version of ARS implements location-based primitives such as isLocatedIn, isAtHome. The work to support other primitives in an intelligent home environment is underway.

Acknowledgement

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

References

[1] Dockhorn Costa, P., Pires, L. F., Sinderen, M., "Architectural Patterns for Context-Aware Services Platforms" in Proceedings of the Second 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] OSGi Consortium, http://www.osgi.org.

[4] Weiser, M., The Computer for the 21st Century, Scientific American, pp. 94-10, September, 1991.

[5] Satyanarayanan, M., "Pervasive Computing: Vision and Challenges". IEEE Personal Communications, pp. 10-17, August 2001.

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

[7] Chen, H., "An Intelligent Broker Architecture for Context-Aware Systems", PhD proposal in Computer Science, University of Maryland, Baltimore County, USA, 2000.

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

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

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

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

[12] Henricksen, K. and Indulska, J., "A software engineering framework for context-aware pervasive computing" in Proc. of the 2nd IEEE Conference on Pervasive Computing and Communications (Percom2004), Orlando USA, 2004, pp 67-77.

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

[14] Philips Ambilight System – http://www.research.philips.com/technologies/syst_softw/ami/ambilight.html

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

[16] Amigo Open Source Repository – http://amigo.gforge.inria.fr/home/index.html

[17] 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.

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

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