Towards an Integration Platform for AmI: A Case Study

Andrés Fortier[1,2], Javier Muñoz[1],
Vicente Pelechano[1], Gustavo Rossi[2,3], Silvia Gordillo[2,4]

[1] DSIC, Universidad Politécnica de Valencia, Valencia, España.
{jmunoz, pele}@dsic.upv.es
[2] LIFIA, Facultad de Informática, UNLP, La Plata, Argentina.
{andres, gustavo, gordillo}@lifia.info.unlp.edu.ar
[3] CONICET
[4] CICPBA

Abstract. The development of intelligent environments involves many different disciplines and skills, which makes unrealistic to think that a single research group or company can develop all the hardware and software required to build such an environment. If we want to effectively support Weiser's vision, what is really needed is an integration platform which allows different components to seamlessly interact, in order to provide pervasive services and ambient intelligence. In this paper we give a first step towards such a platform by presenting a concrete example of integration, which involves two different projects developed at different universities. As a result, we identify a series of problems that we encountered during the integration process and provide knowledge (lessons learned) that can be used by other projects in order to integrate ubiquitous and ambient intelligence systems.

1 Introduction

Creating intelligent environments that support ubiquitous services is a complex task. In this kind of systems knowledge from different disciplines such as HCI, artificial intelligence, software engineering and sensing hardware have to be combined to produce the final application. Therefore, if we want to achieve an ubiquitous environment as the ones envisioned by Weiser [11], it is unrealistic to think that a single research group or company can develop all the required hardware and software. An ambiance of this nature will be conceived as the integration of disparate components developed independently, perhaps with different objectives in mind.

In this paper we argue that what is really needed to support this vision is an integration framework (comprising formalisms, tools and a software platform), which allows different components to seamlessly interact, to provide pervasive services and ambient intelligence. In such framework we should be able to specify, in an abstract way, the contextual information that a certain software module needs to perform his task, so that the integration platform can dynamically discover which providers can fit the module needs.

This paper has been partially supported by the SeCyT under the project PICT 13623 and with the support of MEC under the project DESTINO TIN2004-03534 (cofinanced by FEDER) and FPU grants program.

As an example, consider a user returning to his (intelligent) home: if the home knows about his activity (i.e. driving back home) and can find out what his location is, it can decide to raise the heating thermostat so that when the user arrives the home is warmed as he likes. As a result of this action, a service pops-up in the user's PDA allowing him to remotely vary the home temperature. In this scenario, the home itself can be seen as a consumer of the user's context (namely *where* he is and *what activity* he is performing) and other software modules as the providers of context information. At the same time, the home also acts as a provider, offering to the user a service that manages the home heating system. From our point of view, the home shouldn't care about *who* is providing this information, as long as it meets its requirements; is the integration platform who should be in charge of finding the suitable providers for those requirements.

This kind of ubiquitous system is feasible in terms of existing technology; what seems to be more ambitious is the possibility that the comprising components (e.g. the intelligent home software and the user's system) can be developed independently from each other. As far as we know, there are no specification techniques or discovery mechanisms to support this task. Nevertheless, we do have the bricks and mortar to build them: object interfaces, web services, design by contract, agents and the semantic web are just some examples of available tools.

As a contribution towards the development of an integration platform, we present in this paper a concrete example of systems cooperation. This example involves two different projects developed at different universities. Both projects address the problem of building ubiquitous software, but they do so using somewhat different approaches. The Software Engineering And Pervasive Systems (SEAPS) project, being developed in the OO-Method research group from the UPV, focuses on the development of a model driven method for the automatic generation of pervasive systems from abstract specifications. The systems that are generated by this method are used to provide services in a particular environment, generally smart homes. To implement the systems, the OSGi middleware[6], which is based on Java, was used. On the other hand, the Context-Aware (CA) project being developed at LIFIA, in the UNLP, focuses on the user as a service consumer. In this view of a pervasive system, the user carries a mobile device most of the time, which is used to present services according to his context, which can vary dynamically. This framework is implemented in Smalltalk.

By integrating both systems we expect to achieve three short-termed goals:

- Improve the SEAPS project with dynamically-varying context information.
- Extend the CA project so that it can remotely manipulate SEAPS services.
- Build a context model based on the information sensed by the SEAPS.

We also expect to gain knowledge about more generic integration needs, to be able to effectively build the integration platform mentioned before. As a result of the work carried out, this paper presents the following contributions:

- We show a concrete case of independent systems integration.
- We identify a set of problems encountered during the integration process.
- We provide some knowledge (lessons learned) that can be used by other works in order to integrate ubiquitous and ambient intelligence systems.

2 Background: The Systems to be Integrated

As stated in the introduction, this work presents the integration of two systems that have been developed in the context of different research projects. In this section we briefly describe the main characteristics of the involved systems.

2.1 The Context-Aware Framework

Building applications that provide context-aware services (e.g. those that depend on the user's location) is typically hard task, mainly due to the "organic" way in which they evolve [1]. This characteristic presents a big challenge in terms of maintainability, since the context model and the mechanisms used to sense information change in almost unrelated ways; while context aspects are high-level models of the user environment's features, the sensing mechanisms have to deal with low-level details and are subject to technological changes. To overcome this problem, at LIFIA we devised an architecture that effectively decouples the context model from the sensing mechanisms. To do so, we decompose a context-aware application in two orthogonal views: an application-centered view and a sensing view. The application-centered view, which is organized in three layers, is concerned with the context and service management (a detailed description can be found in [4, 9]). In the **Application Model** layer we specify the application classes with their "standard" behaviors (i.e. those that are not aware of the user's context nor exhibit context-related information).

The **Context Aspects** layer is divided into packages, each one modeling a specific aspect of the user context. Examples of these aspects can be the user's location, network bandwidth, current weather and so on. In this way, instead of creating an application on top of a static context model, context aspects act as software modules that add new behavior to the one provided by the application. As consequence of this model, context aspects can be changed dynamically (even in run-time), creating an implementation close to the notion of context presented by Dourish [3]. When a context aspect changes, the change is propagated by the dependency mechanism to the **Services** layer, which decides if new services should be displayed to the user or if already active services should be removed. Also, the user can decide which services he is interested in by using a subscription mechanisms, so that only those services will be available.

Once the context and service management have been established, we need a mechanism to feed external information into the context model. But, if our context model depends on the sensing details (like data formats, abstraction from low-level data to objects, update frequencies, etc), the resulting application will be hardwired to a specific technology, making it virtually impossible to modify. For this reason we see sensing as a concern that cross-cuts the context layer (see Figure 1) and we claim that there should be a mechanism that hides sensing details to the context model, making it as transparent as possible.

To solve this issue we introduce the *sensing aspect* concept. A sensing aspect is an object that is constantly monitoring a sensor. When the sensor receives a new value, is the responsibility of the sensing aspect to deliver the information obtained to the designated context aspect, performing the required data transformations. In

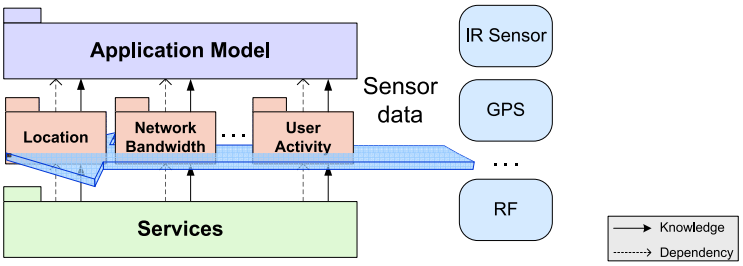

Fig. 1. Schematic View of the Architecture.

this way, the context aspect never has to deal with sensors; in fact it doesn't even know how data is gathered, which means that changes in the sensing view of the system will never impact on the context model.

2.2 The Software Engineering And Pervasive Systems Project

The Software Engineering And Pervasive Systems (SEAPS) project is developed in the OO-Method research group from the UPV. The goal of this project is the development of a model driven method for the automatic generation of pervasive systems from abstract specifications. This method applies the guidelines defined by the Model Driven Architecture (MDA) and the Software Factories. Following these guidelines, the method provides(1) a **modeling language (PervML)** for specifying pervasive systems using conceptual primitives suitable for this domain, (2) an **implementation framework** which provides a common architecture for our systems, and (3) a **transformation engine** which translates the PervML specifications into Java code.

The implementation framework, which is introduced in [8], has been built on top of the OSGi middleware. The architecture of the framework applies the Layers and Model-View-Controller (MVC) architectural patterns [2] for providing a multi-tier architecture for the pervasive systems (see Fig. 2).

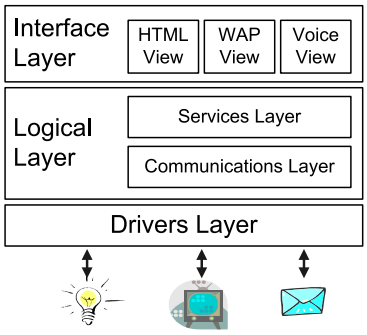

Fig. 2. Architecture of the systems generated by the SEAPS project.

Next we briefly introduce the layers of the architecture:

- The **Drivers Layer** is in charge of managing the access to the devices and to the external software services.
- The **Communications Layer** provides a representation of the devices and the external software applications. This layer holds the manufacturer-independent part of the devices whereas the drivers hold the manufacturer-dependent issues.
- The **Services Layer** provides the functionality as it is required by the users of the system. The components that implement the services make use of the elements in the communications layer or other services in the same layer.
- Finally, the **Interface Layer** manages the access to the system by human or software users. In this layer we make an extensive use of the MVC paradigm, where the components of the Services Layer are seen as the model. A controller element provides functionality for accessing and interacting with the services; this controller is exported as Web Service in order to facilitate the integration with external systems.

This architectural style organizes the system elements in layers with well-defined responsibilities and it provides a clear mechanism (drivers) for the integration with external data or functionality.

3 A Motivating Example

Defining a generic mechanism to provide transparent system integration presents a big challenge. The main reason for this is the lack of an accepted mechanism to define the semantics of a software module, so that automatic discovery and matching can be performed. In an ideal pervasive system, producers would publish their data and functionality, and consumers their requisites; then, the integration platform would be in charge of connecting the matching parts.

As a concrete case, imagine that it is winter and the user is leaving home to get to his office. Based on his agenda, the current time and the fact that he has left the house, the smart home can infer (with an important chance of being right) that the user is going to work and that he won't be coming back until late in the evening. As a result, the smart home decides to lower the heating thermostat to 15 degrees Celsius to save energy.

It is 7 p.m. and the user is driving back home after work. When he reaches a 6 km distance from his home, based on his location, agenda and the current time, the smart home decides to raise the thermostat to 26 degrees, so that when the user arrives, the home is warmed as he likes. Also, the user can see in his PDA that the heating has been raised and has the ability to remotely vary the temperature. Suddenly a friend calls the user and ask him to join with their friends in a casual meeting. By pointing in a map the place when the meeting will be held, the user is able to lower the temperature and re-schedule it to go back to 26 degrees when he leaves to his home.

In the next section we present a description of our experience in integrating the systems previously presented for implementing this case study.

4 Integrating Heterogeneous AmI Systems

To effectively integrate the systems that have been presented in Section 2, we need a communication mechanism that allows both of them to exchange information, taking into account that the systems were written in different languages (Smalltalk and Java) and are based on different architectures. This leads us to considering three main alternatives:

- Using a low-level communication system (like sockets).
- Using a transparent, cross-language distribution framework (like CORBA).
- Using web services to export the desired behavior.

The first option was rejected almost at once, since we should have to deal with a myriad of low-level details (connection, marshaling, request formats), while there was no clear advantage over the other alternatives. Deciding between a distribution framework (such as CORBA) and Web Services is not trivial and the decision cannot be easily generalized.

In the following sections we present our strategy for the three integration goals described in Section 1, explaining in each case the mechanism used to connect both frameworks. We show the limitations prior to the integration and how, based on the motivating example, both systems can be improved. We then give a brief outline of how the integration can be carried out.

4.1 Providing Home Services to External Users

Previous functionality: The CA framework is in charge of providing services to the user, based on his current context. These services must implement a specific interface (named `UserService`) in order to be handled by the framework. Thanks to the distribution framework used (Opentalk), the services can be local or remote, without impacting in the context-aware application. Nevertheless, the services are assumed to be implemented as Smalltalk objects.

Improvement: Integrate the services provided by the SEAPS project to the ones already provided by the CA framework. In the presented example, the temperature control that the user sees in his PDA is an imported home service, which allows him to manage the heating settings of the smart home.

Overall strategy: The CA framework is in charge of modeling the user's context and providing the mechanisms to decide when the services should be presented. The SEAPS system is responsible for providing an external interface to control the devices installed in the house. By extending the notion of a service in the CA framework we are able to plug the exported SEAPS services and treat them as native services. As a result, the imported services are also subject to context conditions, such as being shut down when the user is in an important meeting.

Implementation: This case of integration clearly matches the web services intent, since we are just interested in accessing a specific functionality provided by the smart home. Also, we take the advantage that each home service is already exported as a web service, which is directly accessible to the CA framework by means of an UDDI repository. We also use a publish-subscribe system, implemented as web

services, to inform the user when a new service is available in the repository. In this way the user can decide whether he wants to register to it or not, and under which circumstances the service should be available.

4.2 Extending the User's Context with Home Information

Previous functionality: As explained in Section 2.1, the CA framework provides a mechanism to decouple the context model from the sensing mechanisms, allowing them to change without almost any impact on the other. Thanks to this property, sensors can also be software modules: we can employ the user's agenda or an external web service as a contextual information provider. On the other hand, the SEAPS project has an important infrastructure in terms of sensing devices that could enrich the user's context with home information.

Improvement: Integrate the sensing capabilities found in the SEAPS project to provide information about the home's state. In this way the context model can be extended with home information, providing new adaptation opportunities. In the presented example, the user context is enriched with knowledge about the state of his home appliances, so that when the heating thermostat changes, the user gets a notification and receives the appropriate services.

Overall strategy: The SEAPS system should provide a publish-subscribe mechanism, so that the CA system can subscribe to specific sensors. When a sensor reads a new value, the system should check if the CA system is registered to it. If this is the case, the SEAPS system should send a remote notification to the CA system, indicating which sensor has changed and the new and old values. In the CA system each SEAPS sensor is represented by a proxy, which will receive the remote notification and propagate it as if it where the one which generated it.

Implementation: By combining web services and a publish-subscribe mechanism, we are able to publish each sensor in a repository, so that applications can search them and register as consumers of their events in an individual fashion. In the CA application, a proxy sensor is created for every SEAPS sensor to which it is subscribed. When the remote sensor sends a change event, the local notification web service forwards the event to the corresponding proxy, which propagates the change as if it was originally triggered by him. Thus, from the sensing aspects point of view, there is no difference between a hardware sensor an a SEAPS remote sensor.

Even though the choice of web services over CORBA is not as trivial as the previous example, we find web services better suited for the following reasons:

- Sensors provide a static, well defined set of operations which can be exported as a single entity (there is no need to export an object graph).
- As with context aspects, we aim to have sensors repositories. This matches the UDDI repository intent.
- Web services allow for loosely coupled communication between the parts by having very few information published.

We think that these properties make sensors, exported as web services, good candidates for automatic discovery and replacement.

4.3 Enhancing the SEAP Systems with the User Context

Previous functionality: The smart home systems that are generated by the SEAPS approach must deal only with fixed contextual information: the information that is provided by sensors distributed in the home. This is acceptable since the physical environment (the home) and the required environmental information are both known. Unhopefuly, the current strategy makes difficult to adapt to contextual information, such as the user's preferences or current activity. On the other hand, the CA framework provides a dynamic context model, with rich information about the user's environment.

Improvement: Extend the SEAPS smart home systems for dealing with the user's context. Information about the user's location, activity or music preferences can help to improve the services provided by the home. In the motivating example, the smart home has information about the user's location and his agenda, which, in conjunction with the current time, allows to infer when the user is heading home. Having this information, the home can decide to raise the thermostat's temperature so that the home is warm when the user arrives.

Overall strategy: The integration of the smart home with a context provider can help to improve the current situation. Following this approach, the smart home delegates the acquisition of context information about the users to the context provider. The context provider is in charge of notifying to the smart home about relevant changes in the user context.

Implementation: As in the previous cases, a publish-subscribe mechanism is needed, so the smart home can subscribe to the required context aspect and get a notification when the context changes. In the smart home, this mechanism has been implemented as a driver (see Figure 2) which, instead of providing information directly sensed by a physical device, supplies the information notified by the context provider. When a notification is received, the driver disseminates the information to the services that are interested on it, which could in turn react as a consequence of the context change.

In this integration case, choosing between web services and a distribution framework is not easy at all. In Table 1 we present a brief summary of the pros and cons of using each option for implementing this case.

Based on these arguments, in this particular example we choose to integrate the systems using web services, because the context aspects don't require behavior distribution and we have more experience on using web services.

It should be noticed that this doesn't mean that we have discarded distribution frameworks like CORBA. This first case study has presented an important dichotomy in terms of whether web services or distribution frameworks should be used. In the near future we expect to identify which context aspects should be exported as web services and which ones with a distribution framework.

Web Services	CORBA
(×) Not well suited for distributing complex object models.	(√) The whole object oriented model can be accessed.
(√) Promote loosely coupled systems.	(×) Promotes tight coupling.
(√) Publish context aspects in distributed repositories facilitates scalability.	(×) Firewall traversal problem.
(×) Adapting a dynamic language as Smalltalk to Web Services interfaces is a cumbersome task.	(×) Adapting a dynamic language as Smalltalk to CORBA interfaces is a cumbersome task.
(√) The idea of independent web services providing a specific functionality exactly fits the notion of a context aspect.	
(√) Are a mainstream in the industry.	

Table 1. Web Services and CORBA pros and cons.

5 Challenges Identified

The development of this work have raised several questions that can be generalized for any integration project in the context of Ambient Intelligence systems. The main challenges that we have identified are:

- In AmI systems, the integration of services and the contextual information must be dynamic, automatic and seamless. The challenge is: **how do we specify and publish the contextual information requirements in a way that these requirements could be satisfied by third-party and previously unknown context providers?**
- The previous challenge also applies to sensors and services. For example, we would like to ask for a sensor that provides the user's position in a symbolic notation without caring if this information is obtained through a GPS in the user's PDA or by an external tagging mechanism.
- Currently, only contextual data can be shared in heterogeneous environments. The challenge is: **how do we share not only data but also behavior, in a decoupled way?** Distribution frameworks provide the ability to share behavior, but the price to pay is tight coupling between the involved parts; web services solve this problem by publishing a very small interface, but as mentioned before, they are not well suited to share object's behavior.

6 Lessons Learned and Future Work

In this paper we have introduced a practical case of integration in the context of AmI Systems. Starting from independently developed systems which face different AmI aspects, we have proposed and implemented three integration goals. As a result, four important lessons have been learned:

1. An integration platform is needed to provide dynamic AmI environments.
2. The platform should be able to handle three main abstraction concepts, namely: *sensors*, *services* and *context aspects*.
3. The notion of independent, loosely-coupled software modules clearly matches the Web Service's intent. Both sensors and services adapt extremely well to this metaphor and can be implemented straight forward.
4. The subscription/notification mechanism has been widely used to connect both systems. This is not a surprise, since this communication paradigm provides not only a clear decoupling between the message provider and consumer, but also a decoupling in time, due to it's asynchronous nature.

As future work we plan to explore the following research lines:

- Port our ad-hoc publish-subscribe system so that it complies with emerging standards as WS-Eventing [7] or WS-Notification [5].
- Investigate the third version of the UDDI specification, since it already solves the notification issue [10].
- Further investigate if the three proposed abstractions (sensors, services and context aspect) are enough to build most (if not all) AmI scenarios.
- Deeply **analyze other research areas which face related problems** like the intercommunication of agents and the semantic search of web services. Works in these areas point out solutions like the use of logic descriptions or reference ontologies for enriching the data description.

References

1. Gregory D. Abowd. Software engineering issues for ubiquitous computing. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 75–84, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
3. Paul Dourish. What we talk about when we talk about context. *Personal and Ubiquitous Computing*, 8(1):19–30, 2004.
4. Andres Fortier, Gustavo Rossi, and Silvia Gordillo. Decoupling design concerns in location-aware services. In *Mobile Information Systems II*, pages 187–202, 2005.
5. IBM and Akamai et al. Web services notification (ws-notification). http://www-128.ibm.com/developerworks/library/specification/ws-notification/.
6. Dave Marples and Peter Kriens. The Open Services Gateway Initiative: An Introductory Overview. *IEEE Communications Magazine*, 39(12):110–114, 2001.
7. Microsoft, IBM, and BEA. Web services eventing. http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf.
8. Javier Muñoz and Vicente Pelechano. Applying Software Factories to Pervasive Systems: A Platform Specific Framework. In *8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Paphos (Cyprus), May 2006.
9. Gustavo Rossi, Silvia E. Gordillo, and Andres Fortier. Seamless engineering of location-aware services. In *OTM Workshops*, pages 176–185, 2005.
10. Uddi v3. http://uddi.org/pubs/uddi_v3.htm.
11. Mark Weiser and John Seely Brown. The coming age of calm technolgy. *Beyond calculation: the next fifty years*, pages 75–85, 1997.