We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

2

Services Everywhere: an Object-Oriented Distributed Platform to Support Pervasive Access to HW and SW Objects in Ambient Intelligence Environments[1]

Jesús Barba, Félix Jesús Villanueva, David Villa, Francisco Moya, Fernando Rincón, Maria José Santofimia and Juan Carlos López
University of Castilla-La Mancha
Spain

1. Introduction

The Ubiquitous Computing concept was first defined by Mark Weiser (Weiser, 1995) and it refers to a new computing era where electronic devices merge with the background. People make use of those electronic devices unconsciously, focusing just on their needs and not in how to accomplish them.

The concept of Ambient Intelligence (Ducatel et al., 2001), lying on the ubiquitous computing paradigm, refers to those environments where people are surrounded by all kind of intelligent intuitive devices, capable of recognising and responding to their changing needs. People perceive the surroundings as a service provider that satisfies their needs or inquiries in a seamless, unobstrusive and invisible way.

It is generally agreed that AmI (Ambient Intelligence) will have a great impact in economy and society. The potential of AmI technologies in various application areas has been object of numerous studies. For example, the *IPTS/ESTO Ambient Intelligence in Everyday Life Roadmap* (Friedewald & Da Costa, 2003) report analyzes key application areas in order to find out which are the technological requirements that must support the functions that will make the difference in each of those application areas (housing, mobility and transport, shopping and commerce, education and learning, health, culture, leisure and entertainment).

But AmI is not only its technological facet. There is also a social and a politics dimension besides the devices and software upon which the intelligent environments are built. Technology must be helpfull, work with users and not against them trying to pull down the wall of the natual resistance of the human being to revolutionary changes.

[1] This work has been partially funded by the Spanish Ministry of Science and Innovation under the project DAMA (TEC2008-06553/TEC), the Regional Government of Castilla-La Mancha under project RGRID (PAI08-0234-8083) and the Spanish Ministry of Industry under project CENIT Hesperia.

Putting the focus on the technological dimmension, the requirements for AmI to become real are varied and can be classified in two domains or computational areas (ISTAG, 2003): technologies for *ambience* and *intelligence*. An intensive reasearch in each one of them, by its own, is not a guarantee of success but an additional effort in finding the mechanisms to ensure the integration of components and devices in AmI systems must be done. Moreover, such integration must be perfomed in a seamless way.

Therefore, the importance that the *technologies for integration* play in AmI is out of any doubt. We strongly believe that the different research challenges in this area are in the core of the success of AmI. However, currently, many of the AmI demands for a hardware platform and software architecture, suitable for its needs, have not got a satisfactory response yet. The ISTAG in 2003 identified the following features that a platform for AmI must include: *abstraction, automatic composition, interaction management, computational efficiency, creativity, scalability and evolution* and *dependability*.

It can be easily checked that we are sill far from such vision since AmI environments are currently limited to a few demonstrators in the research centres, and the incorporation of the AmI technology to real scenarios and applications is taking place extremely slowly. Next, we try to be more precise and analyze in deep, from our point of view, the main reasons of such delay in the realization of AmI.

Within the layered vision of an AmI system, the lowest level is in charge of collecting data that will be translated into contextual information by upper levels. This layer is mainly composed by sensors, small devices with limited capabilities (memory and computational power). At this level, it can also be found a set of actuators that execute the commands coming from the layers that generate decisions from the data gathered by the sensors. On top of the *sensors & actuators layer*, a set of nodes with sufficient computing capabilities interpret the raw data, extracting the knowledge contained in them (in an automatic way). Moreover, the nodes of the upper levels may provide value-added services like the collection of statistical data or the integration with business processes.

So, in the basis of any AmI application there is a *complex, extremely large distributed system* containing not only nodes with uneven features regarding execution speed, battery life or interfaces but also a *communication infrastructure* that surrounds them and probably combines several *communication protocols* (i.e. RFID, Wi-Fi, WiMax, Bluetooh, etc). Such mixture of properties conform a *heterogeneous platform* that is intended to support the delivery of value added services to software components that are built on top of it.

It is difficult to tackle with heterogeneity since it poses additional difficulties to the development of AmI platforms and applications with the characteristics previously mentioned. First of all, programmers should be aware of such heterogeneity, inherent in AmI systems, and deal with several languages, programming interfaces, network protocols, etc. The time invested by the developers in these affairs is time they are not using to implement the *intelligence* of the nodes that, in any case, is what makes the difference. Also, the productivity of programmers will increase since they do not have to worry about the implementation details of the integration and communication layers.

To sum up, it is ***mandatory in AmI environments a tailored middleware platform to help the application and service developers with the integration of components in a simplified and seamless way***. Such middleware platform must not only provide the necessary semantic to connect and make the different elements in the system interoperable (acting as the glue between them) but also must offer a set of tools, methods and even embedded

services (i.e. composition and reconfiguration services) to enable the rapid exploration and prototyping of new systems or more complex services.
Although AmI environments have much in common with distributed objects platforms developed in the nineties (i.e. CORBA, EJB, DCOM …), and also with current grid computing platforms (i.e. Globus, gLite …), such distributed platforms where not originally conceived thinking of the special needs of AmI. Therefore any attempt to extend standard middlewares and apply them to specific AmI applications or scenarios must be carried out carefully, trying not to inherit the deficiencies of the original middleware concerning AmI. On the other hand, such adaptation process represents an extra effort in order to make the former middleware converge with the unforeseen requirements imposed by the new scenario. Also, the development of new services and communication means for the standard middleware is not for free and, probably, their reuse in other AmI applications can be hard to accomplish.
To overcome the limitations of current approaches, the ARCO research group at the University of Castilla-La Mancha[2] started to work on a global solution, specifically conceived for AmI environments, based on its wide experience in distributed object platforms and application oriented middlewares[3].
Our proposal leverages the achievements of standard middlewares in regular networking distributed platforms and comprises:

- A standard communication platform which allows developers to integrate heterogeneous devices and networks.
- A common development framework to build new services or easily integrate legacy ones.
- A set of services that provide the developers with advanced features such as service discovery, positioning, management of reconfigurable hardware, migration, event channels, etc.

Throughout the rest of this section we sketch the main features and advantages of our work and how it effectively answers many of the major challenges in platform and software design for AmI as stated by the ISTAG.

1.1 A holistic approach to the major AmI challenges

The main objective of the Object-Oriented distributed Platform for AmI (from now on OOPAmI) is to address the demands of the emerging AmI technologies from an *integrated point of view, based on the Object-Oriented distributed paradigm*. Next, we detail how this primary goal is achieved:

- *Abstraction.* Both services and devices are modelled as objects. The concept of object *effectively abstracts* the hardware and network technology used to deployed the services in AmI applications. This allows service developers to concentrate their effort in achieving better and more intelligent algorithms instead of worrying about communication and platform dependent problems.

[2] http://arco.esi.uclm.es
[3] Among the projects handled by the ARCO research group covering these topics it is worthy mentioning those regarding domotic, homeland security and ubiquitous & pervasive computing.

- *Automatic composition*. On top of the object communication engine, a set of *Intelligent Agents* implement a reasoning engine which main responsability consists in responding to incomming events, needs or requirements by means of service composition. The so called, *Multi-Agent Service Composer System* is implemented using the Jadex platform.
- *Computational Efficiency*. OOPAmI is able to integrate in the platform architecture: (1) low footprint devices such as eight bit microcontrollers (micro-components); (2) full servers or mobile devices (i.e. PDAs, SmartPhones) with enough power of computation to run complex tasks or algorithms; and (3) application specific hardware in form of reconfigurable logic or Systems-on-Chip. The variety of the devices that OOPAmI can handle makes our platform flexible and easy to adapt, depending on the level of performance demanded by the target application. The transparent integration of fully customized hardware devices allows OOPAmI to cope with the most complex tasks. Besides this, the use of low cost, small computing devices for the simplest services (i.e. retrieving data from sensors or the control of the actuators) helps to achieve the maximum resource usage efficiency, lowering the final cost of the system.
- *Scalability and Evolution*. OOPAmI defines and implements a migration service. Such migration can take place from: (1) software nodes to software nodes (with different computing capabilities); (2) software nodes to hardware nodes and vice versa. The migration service facilitates the management of a punctual increment of the work load in the system, enabling the use of the whole available resources in the system. Once again, a set of *Intelligent Agents* are in charge of reasoning the better action in each scenario in order to provide the desired degree of quality of service. The *Multi-Agent Migration Service* uses the above mentioned migration service and a reconfiguration service to deploy the bitstreams on the reconfigurable logic devices where necessary. New hardware designs and algorithms can be incorporate to the system anytime.
- *Dependability*. Run-time failure management and security are also considered in OOPAmI. Transient and permanent errors in electronic devices are handled using different approaches: (1) utilization of replicas; (2) replacement of the affected node (i.e. downloading a new bitstream that fixes the malfunction); and (3) movement of the objects that run on the broken node. In OOPAmI these three scenarios are transparent to the developers due to: (a) a special distributed location service with extended capabilities; (b) a failure detection and notification service; and (c) a persistence service which is used by the objects to periodically save its internal state in order to be retrieved latter in case of failure.
- *Software and Service Architecture.* OOPAmI makes it use of the *Distributed Object Based Services* (DOBS)[4] that comprises several research projects within the ARCO group. The first implementation of DOBS targeted home services and, after several iterations it has evolved into a development and integration framework for AmI services. The Multi-Agent Service Composer System relies on DOBS and a OWL ontology to perform automatic composition of services.

[4] http://arco.esi.uclm.es/dobs

- *Design.* A unified design flow for DOBS objects is proposed. DOBS objects can be implemented on Wireless Sensor Network devices, mobile devices, computers or hardware.
- *Integration*. From an integration point of view, objects unify the way services are accessed which leads to better and low effort integration mechanisms.
- *Business model*. Nowadays, the role of different companies in the digital home market (the prelude of real AmI environments) is reduced to the engineering of costly projects regarding the control of *smart building*. Such projects are mainly based on the concept of *residential gateway* which is totally opposite to the openness principle demanded by AmI. It is important for AmI projection to open the market to new actors establishing open architectures and protocols. Roles as "service providers" or "resource providers" may then be possible so that a competitive market can be established. In this line, OOPAmI is a unified middleware *able to interact with standard middlewares* (i.e. CORBA, Internet Communication Engine) which facilitates the integration of new services or devices from any vendor.

To help the reader to visualize and understand the numerous contributions of OOPAmI, Table 1 summarizes its most relevant features and their relation with the requirements for the best AmI platform.

OOPAmI platform features and components	**Application to AmI environments**
• "Object" as the modelling concept	• Abstracts the platform • System-level focus • Helps to cope with heterogeneity • Offers the necessary semantics to integrate services and devices • Reusability and interoperability
• Multi-Agent Service Composer System	• Automatic composition of services
• Reconfigurable nodes	• Easy management of adaptive systems • Migration • Performance and flexibility
• Event channels	• Platform-independent data communication and acquisition
• Distributed Object Oriented programming model	• Single programming model • Easy development of algorithms and applications • Homogeneous view of the system • Automatic generation of embedded code

OOPAmI platform features and components	Application to AmI environments
• Low cost nodes (microcontrollers)	• Flexibility • Maximum resource usage efficency • Computational efficiency
• Multi-Agent Migration Service	• Scalability, evolution and dependability
• Distributed Object Based Services	• Automatic composition of services • Easy devlopment of services
• Unified design flow	• Design of services, hardware and embedded software
• Interaction with standard middlewares	• Openness

Table 1. Main OOPAmI features and their application to AmI platforms.

2. Previous work in middlewares for AmI

Due to the huge number of knowledge fields that AmI includes, there are many previous works related with AmI hot topics (integrated circuits, artificial intelligence, etc.). However, in this section we will only focus on those works regarding the middleware technology used to overcome some of the problems in AmI environments.

The middleware has an increasing number of tasks associated in AmI environment, and its design has to be carefully delimited to cope with all of them. Between these tasks we can mention homogeneous access to AmI devices and resources, with independence of the location, operating system, technology used, language used in the implementation, etc. of the services.

Maybe one of the key points of the approach is the interoperability between different technologies and services. This problem is not resolved yet. In the meanwhile some technologies are called to play an important role in the AmI environments, and therefore their integration must be considered:

- Radio Frequency Identification (RFID) starts to be a predominant technology on the identification field and all related applications (e.g. vehicles fleet control, goods tracing). RFID is also associated with applications that require basic information storage (i.e. identification cards). RFID is becoming indispensable in applications for user interaction. Usually, the RFID tag has scarce resources and existing middlewares can only be supported by the reader (generally attached to a desktop computer).
- Wireless Sensor/Actuator Networking (WSANS) is also another key technology in the future of AmI environments. AmI requirements of monitorization and control of the real environment makes use of WSANs as the interface to the real world.
- Body embedded devices. Currently, there is significant research in devices attached to clothes, or even directly to the human body. Going beyond of body area network, these devices supply data to the environment such as people health

indicators, preferences, and even emotional state that could hardly be extracted with any other technologies (for example, video analysis).

- Vehicular Ad-hoc Networks (VANET) is another emerging field to integrate in the AmI environment since the car constitutes our main mean of transport work and it is part of our daily live.

Some approaches that try to solve the problem of heterogeneity have chosen the java language (for being a multiplatform language) (Sacchetti et al, 2005) or XML and Web Services for the same reason (Issarny et al, 2005), (Perumal et al, 2008), etc. However, web services impose stronger requirements to the devices because they need to support a set of protocols that some of the previously mentioned technologies are unable to support. In the same way, Java requires its java virtual machine that, even in its more reduced version, is too big for devices like wireless sensor networks devices, body embedded devices, etc.

Once again, we must highlight that all the technologies mentioned before (but VANET) encompass devices with scare resources. This is why seamless integration of these technologies requires a light middleware (see section 3.1).

On the other hand, a middleware for AmI environments must be enhanced with extra responsibilities. It must provide advanced services and features to the developers, such as the ones listed below:

- People and resource positioning services: several AmI scenarios require knowing people location, in order to adapt its functionality.
- Automatic service composition: in a real AmI environment, the middleware should compose services that are hard to predict.
- Dynamic discovery of new devices and services: An AmI environment should be dynamically built upon the spontaneous appearance of devices and services., in order to put them all to work together.
- Developer assistance. For a better acceptance of any middleware technology, a simplified design flow based on a complete toolchain is recommended. One example of a very important facility a middleware should provide is a simulation platform. This simulation platform should emulate the devices, services, people and the information flow of a real AmI environment. There are some works with this idea in its aim, like for example (Maly et al., 2008), but usually they are not associated to any middleware so the developers can only emulate standalone algorithm with no interaction with emulated devices and services.

Most of the middlewares developed for AmI environments have not considered these and other aspects (as those enumerated in section 1.1).

The IST Amigo (Sacchetti et al., 2005) project has developed a Java middleware based on the OSGi platform (OSGi Alliance, 2006), extending the concept of residential gateway[5]. IST Amigo middleware also supports the development of services using HTTP/SOAP remote procedure calls. In our opinion, the role of a residential gateway has been encouraged by telecommunication companies with the intention of promoting the client loyalty and extending the service offer. The residential gateway approach has been shifted due the success of other devices that can also play the role of residential gateway (i.e. mobile phones, TV, etc.). The vision of a centralized point for service has definitely changed.

[5]A residential gateway is a single device that interconnects all networks (external and internal) and devices in the environment.

As we argued before, supporting the Java Virtual Machine associated to JAVA, or the protocol stack for HTTP/SOAP, imposes strong requirements to the devices that can be integrated using a middleware. The middleware developed in the IST Ozone project (Gelissen, 2005) is also based on web services developed in JAVA, sharing the same problems than Amigo Middleware.

These web services-based middlewares use WSDL (*Web Service Description Language*), an XML based language for service description. Nevertheless, although XML was designed with human readability in mind, a non trivial service description is hard to understand and use from a developer point of view. Service descriptions should be a natural action for service developers, not a maze of syntactical artifacts (as it happens with WSDL). With this principle in mind, our vision about service description simplifies WSDL based approaches because: (1) we use a simpler language (IDL) for service description; (2) we decouple the description of the interface from its attributes, simplifying the definition of complex devices; (3) we provide tools for attribute modelling.

Agent oriented middlewares constitute another approach for AmI. The works presented in (Wu, 2008) and (Marsa, 2006) are an exponent of this approach. In this type of works, the authors generally try to model human behaviour, or apply artificial intelligence techniques to AmI environments (Ramos et al, 2008) by means of software components called agents (they are typically programmed using the Java language for agent implementation due its possibility of serialization).

We have performed a short travel around what is done in the state-of the art in middlewares for AmI. Many of them only propose partial solutions to some facets of the problem as service composition, network interoperability or service discovery. Others do not even consider important features demanded in AmI middlewares (integration of small devices). We are looking for an integral approach for AmI development as we will describe in the following sections.

3. Integration of heterogeneous devices in OOPAmI

In this section we are going to describe how OOPAmI addresses the heterogeneity problem in AmI environments. One of the most remarkable features of our platform is the capability of integrating devices of different nature based on the *distribute object paradigm*. As justified in the introduction of this chapter, the concept of object provides the necessary semantic in AmI environments to get interoperable heterogeneous nodes due to its capability to abstract the implementation details of the network.

This approach is already present in many standard distributed object middlewares and it has been proved to be useful in the design of new and challenging applications for ubiquitous computing and ambient intelligence environments.

Nonetheless, embedding standard object middlewares require too much computing resources in the target devices in order to implement the whole middleware protocol features. The strict requirements imposed by this approach limit the kind of devices that can be incorporated to the AmI network. For example, the facto obligation of an operating system (i.e. to provide the access to the network interface) knocks out small devices with enough capabilities to implement the object or service functionality. Current solutions do not concern this matter and propose the utilization of oversized devices for the

implementation of simple nodes where most of the resources are assigned to the execution of OS routines or unnecessary middleware services.
Conversely, OOPAmI allows the incorporation of low cost devices to the platform[6], removing the need of the OS and the middleware burden in the nodes without sacrificing interoperability.
The principle behind this revolutionary approach can be summarized in the following sentence: *"Although it is important that each device looks like a distributed object, it is not essential that they are actual distributed objects. If devices are able to generate coherent replies when they receive redefined request messages then the system will work as expected"*. (Villanueva et al. 2007). Such principle is also applied in OOPAmI to another special kind of nodes called *hardware objects*. A hardware object (HwO) is a custom integrated circuit that will perform a complex task or implement a service.
The shift in the utilization paradigm of dedicated hardware in OOPAmI, is considering such custom devices as *autonomous* entities in the network. Traditional approaches, view dedicated hardware as slave accelerator units under the supervision of a master controller (typically a microprocessor). Once again, the complexity of the infrastructure surrounding the special hardware unit is almost entirely given to run the OS and middleware services to provide interoperability. Thus, the cost of the supporting platform rises and so does the cost of programming the embedded software.
To avoid the dependence of custom hardware in a processor-based computation node, we have ported the kernel of a communication middleware onto a pure hardware implementation. The resulting communication infrastructure eases the integration of HwOs within the AmI network since they are able to understand the protocol messages of the underlying middleware.
The remainder of this section is dedicated to explain the architecture, tools and methods that enable small devices and custom hardware nodes to be incorporated in AmI application in a seamless way. Thus, the resulting middleware platform that sustains the services above the integration level is endowed with unforeseen features and devices in other commercial solutions. To exemplify the concepts, methods, tools and prototypes developed to this end we have chosen ZeroC ICE (Henning & Spruiell, 2008), an excellent CORBA like middleware, but the same approach is also applicable to other middlewares.

3.1 Ultra low-cost nodes in AmI platforms

When a node in the AmI network just holds an application-specific server (i.e. read a magnitude value from a sensor), the service, the whole communication engine and its API can be handled by a special object implementation called *picoObject*. A picoObject lacks in a local communication engine and it does not need of object adapters, marshalling routines, etc. We just need to implement the message handling code for the middleware protocol messages whose destination is an object placed at the device.
Nonetheless, for the rest of the network a picoObject behaves as a usual object. It provides a network level interface without significant differences with respect to a standard object. In our case, picoObjects are fully compliant with *ICE protocol*, the network level contract in ZeroC ICE.

[6] Since miniaturisation is a must to accomplish with the *very unobtrusive hardware* principle, is mandatory to offer an integration means for low cost devices with a minimal footprint.

PicoObjects can also handle client-side communications using similar techniques. Client-side messages are composed as a set of templates with just the bare minimum configurable fields.

We have identified a minimum set of rules that must be followed in the development of picoObjects in any target platform:

- Always be compliant with the standard message format for the communication protocol.
- Only offer support to the simplest protocol version whenever interoperability is not compromised.
- Do not offer support for common middleware services (e.g. Naming and Event services), delegate such responsibilities to other nodes in the network.
- Always use a fully static implementation.
- Resident objects are always on. There is no way to activate or deactivate objects.

The simplest way to achieve a coherent behaviour for each picoObject is by means of a message matching automata. In this context, the allowed set of messages that a certain object understand constitute a BNF grammar defined by the following elements: (1) the message format for the middleware communication protocol; (2) the identity of the object (a unique identifier which is unique in the network); (3) the interfaces exported by the object. It includes the signatures of the provided methods (name, arguments and return); (4) the interfaces that must be inherited from the communication engine and implemented (i.e. Ice::Object is case of ICE); (5) the data serialization rules; and (6) the constraints for the target platform.

Using this information, we are able to automatically generate a fully functional parser whose mission is to identify a whole request message. The corresponding user procedure is automatically invoked and a reply message is generated when a matching happens. If the parser fails to identify a valid method request, then the message is discarded.

Input and output messages are handled on-the-fly using a generated byte-stream processor which saves memory since the incoming message has not even need cached by the device. The request message is processed as the bytes arrive and the reply message is also generated partially from replication of the incoming data. The last part of the reply message (the return value) is generated by the user procedure for each method.

As a proof of the feasibility of our approach, we have developed several prototypes of the picoObjects for two existing middlewares: the already mentioned ZeroC ICE and CORBA. The collection of target platforms and languages used is varied and ranges from assembler for Microchip PICs, Java on a standard embedded PC, Java on an embedded Dallas Semiconductors TINI device, C on a standard embedded PC, C for a Zigbee CC2420 platform, etc.

Table 2 shows an extensive comparative between our picoObjects with equivalent implementations in several commercial platforms. It is important to highlight that our picoObject approach results into solutions that are two orders of magnitude (considering the OS and libraries even more) smaller than their counterparts using commercial solutions.

Middleware	Client	Server (node)	Other	Platform
nORB	-	509000	Libs + OS	PC / Linux
UIC	29000	35000	Libs + OS	WinCE / SH3
UIC	16000	-	Libs + OS	Palm
LegORB	6000	-	Libs + OS	Palm
MQC	14590	22110	JVM + OS	TINI
UORB	45000	45000	JVM + OS	Unknown
Maté	-	16044	OS	MICA / TinyOS
TinyDB	-	58000	OS	MICA / TinyOS
TinyLime	-	16000	OS	MICA / TinyOS
SensorWare	-	237000	OS	IPAQ / Linux
Impala	-	≈ 18000	-	Zebranet
WSP	-	27278	-	Unknown
picoObjects				
ICE	2 914	3092	-	PIC16F690 / asm
CORBA	2 640	2962	-	PIC16F690 / asm
ICE	-	≈ 6000	OS	PC / Linux / C
CORBA	-	≈ 5000	OS	PC /Linux / Java
CORBA	-	≈ 4096	JVM + OS	TINI /Java
CORBA	-	≈ 5500	OS	PC / Linux / C

Table 2. Minimum server and client sizes on embedded middlewares (the values are expressed in bytes).

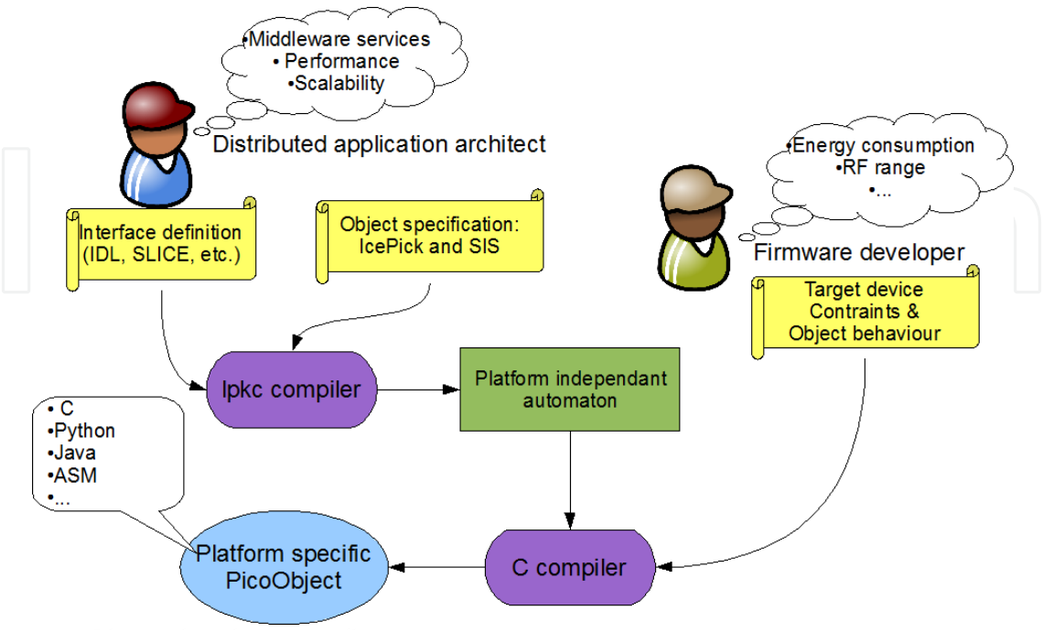


Fig. 1. PicoObject development flow.

The development of picoObjects is supported on OOPAmI by a set of tools and languages that can be used as facilitators by the application developers and embedded code programmers. Figure 1 depicts the whole development flow of picoObjects and the dependencies between the different actors in this process.
First, the distributed application architect must specify the objects present in the system and their relations. Such specification does not only refer to the objects that are going to be implemented as picoObjects in the device but also to those objects that have any relation with them. An interface description language such as IDL or SLICE is used to this end. Along with the operations supported by the objects, it is necessary to provide the object identifiers and the type of relation existent between the picoObject and other objects. For example, a relation cause-effect (i.e. when a picoObject receive a message of type T from X, then reply with a message of type M) or temporal relations (i.e. timers, send periodically a message of type T which is useful for announcement purposes). The IcePick language has been specifically defined for this purpose.
Then, the *lpkc* compiler collects all this information and produces a platform independent definition of the automata (FSM bytecode) that is able to interpret the middleware messages relevant to a picoObject. The automaton, the device restrictions and the implementation of the objet behaviour (provided by the firmware developer) are the inputs for a compiler written in C which produces the final implementation of the picoObject.

3.2 Custom hardware nodes in AmI platform

As we introduced in the beginning of this section, we have versioned an implementation of the system-level middleware entirely in VHDL (Barba et al., 2007). The very first consequence of this fact is that the integration of pure hardware actors[7] in AmI environments is performed in a seamless way.
In the various demonstrators we have built to prove the transparent communication between our hardware nodes and external components, we take as the reference the ZeroC ICE object-oriented commercial middleware widely used in the industry.
The base architecture of a hardware node in OOPAmI is shown in Figure X and comprises:

- The *objects*, implemented as hardware units. A single node can group several of this HwOs, enabling resource sharing and thus, reducing the cost of the final implementation. HwOs can be implemented as static objects or dynamic objects using reconfigurable logic. Dynamic objects can be instantiate and evicted into the reconfigurable area at run time. All the tasks concerning the management of the reconfiguration process are performed by the *Reconfiguration* of the middleware.
- The *External Object Adapter (EOA)*. It provides connectivity with external objects and/or systems. One side of the EOA depends on the component that acts as the bridge with the external network (Remote Network Interface) so it has to be hand made. However, on the side interacting with the local on-chip network (if it is present), the control logic is fully customizable and generated in an automatic way.
- The *interconnection fabric*. When there is more than one HwO embedded in the node, the in-chip communication infrastructure allows the connection of the HwOs

[7] The presence of a processor, running the operating system or a software controller routine is optional.

with the EOA (if not, such connection is done through a point to point link). Also, the HwOs may use the bus, Network-on-Chip or any communication architecture present inside the chip to interact.

In other approaches, since a common communication infrastructure is missing, on-chip functionality may only be accessed from off-chip components using an ad-hoc interface that exists only if it has been predicted by the designer.

The External Object Adapter

The EOA has mainly two duties, namely: 1) translates the ICE object identifiers and operations (strings) into internal local addresses; and 2) adapts the ICEP (ICE protocol) message format to the in-chip message format. Figure 2 represents the place that the EOA occupies in the hardware node architecture. Next, we detail the role of each functional unit using the two possible communication scenarios:

- Incoming message treatment. The *packet analyzer* examines the incoming frames and throws away those that are not valid ICEP messages. The UDP/TCP listening ports and the allowed sources (Internet Protocol addresses) can be configured in the EOA. The *packet sequencer* module internally caches some parts of the ICEP message using the *in-progress request info memory* in order to build a later response message. Finally, the packet sequencer injects the message body "as is" without pre-processing the data because the total compatibility of the coding rules (as we detail below). The target HwO is addresses using the information maintained by the *external routing table* (ERT), the packet analyzer uses it to translate the ICE identifiers.

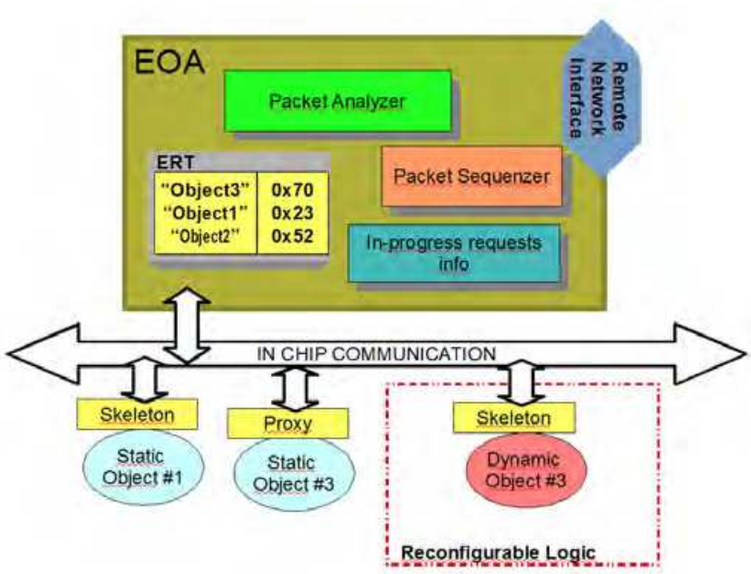


Fig. 2. OOPAmI hardware node architecture.

- Outgoing message treatment. When a in-chip transaction is addressed to an object that is not implemented in the hardware node (i.e. "Object2" in figure 2) the EOA routes the traffic to the external network. The *sequencer* fills a frame

header template (it depends on the external network packet format) with the valid ICE identifiers, external network address, etc. The needed information can be retrieve from the ERT (for a request) or the data previously cached (for a response). Finally, the frame is sent throw the remote network interface.

Hardware Objects (HwOs)

A HwO is the logic that implements the functionality of the equivalent software object plus the logic of the wrappers (the hardware version of classic *proxies* and *skeletons*) that are in charge of translating in-chip transactions in *local invocations* to the HwO. We intentionally decoupled communication from behaviour (following the *Remote Method Invocation* semantics) to make HwO modules reusable in future designs. By isolating HwOs from communication implementation details we make them immune to unforeseen changes in the communication infrastructure.

Both proxies and skeletons agree in how method invocations translate in a sequence of low level actions (write and read primitives[8]) over the interconnection architecture to activate the execution of the operation. Actually, at the logical level a method call results into an exchange of message between the initiator and the target (HwOs can be either initiators or targets of a method invocation).

The RMI protocol for hardware implementations defines the format of the message to be sent and how to code the arguments of both invocations and results. For example, the header fields of a message are combined to form the address of the destination of a transaction.

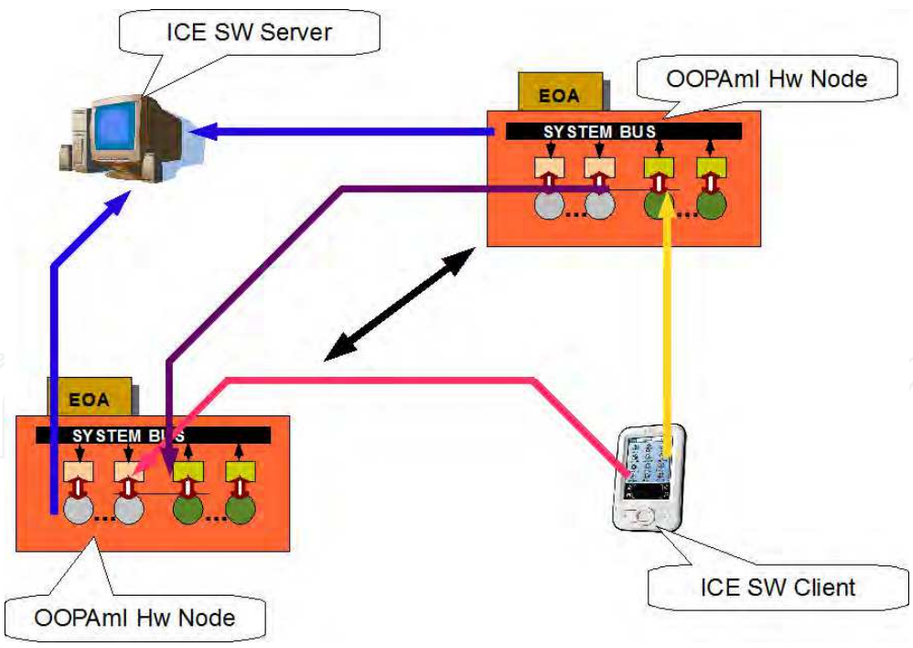


Fig. 3. Hardware and software node integration in OOPAmI.

[8] They are basic services offered by most of the buses so that we do not limit the platform implementation to a concrete technology.

But what really provides the inter-component communication semantics with standard ICE software clients or server as well as with other hardware nodes is the use of the same data type system and coding rules. HwOs support all basic types (bool, short, int, float, etc.) plus structures, sequences (vectors) of a fixed size and any combination of them. This makes our hardware implementation of the middleware 100% compatible with a well known subset of ICE features. The encoding rules defined by ICE are quite simple in order to propagate such simplicity to the architecture elements that will manage the marshalling and unmarshalling processes (proxies and skeletons). This simplicity allows reaching an efficient component design with the minimum overhead.
Thus, the format of the body messages remains unchanged, no matter the nature of the communicating objects, which means that a target HwO is not able to distinguish whether the source of the invocation is a SW or another HwO.
In Figure 3, four interaction scenarios are depicted involving hardware software nodes. It is shown how existing on-chip objects can communicate with SW servers (blue line) and server objects implemented (black and purple lines) as HwOs. Also, HwOs can be accessible from outside (yellow and red lines).

Development flow
Along with the definition of an ICE compatible hardware platform architecture, we have developed a design flow to easily create and integrate HwOs. Figure 4 shows the workflow of the proposed HwO generation process. The input to this process is an object interface description, written in an interface description language (SLICE in the case of ICE), just like the other objects will see the hardware component in the system. This information is used to generate: (1) the on-chip communication infrastructure and (2) the final hardware object implementation.

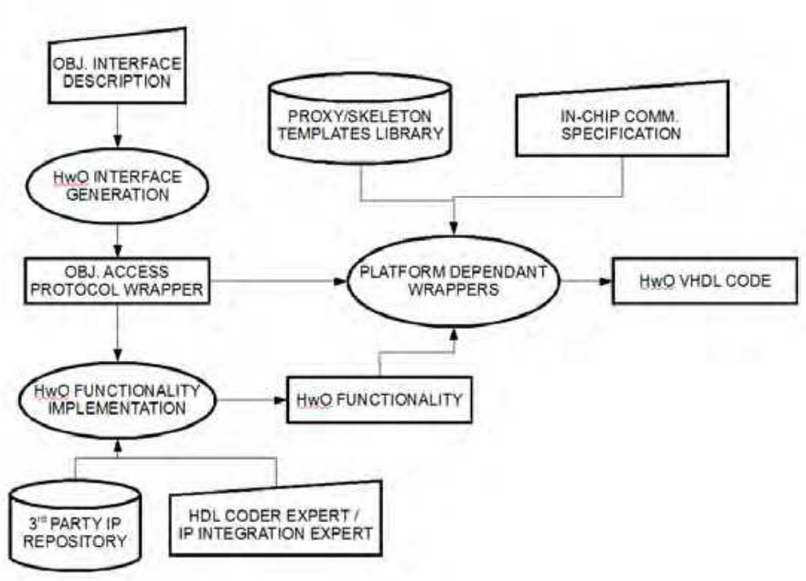


Fig. 4. Design flow of HwOs in OOPAmI

This HwO implementation can be the result of an integration process, reusing existing designs implementing the required functionality. A new, hand coded chip implementation is needed when no reuse opportunities are present for the current application.
Since we have standardized the way the functionality of a HwO is invoked (HAP, the *HwO Access Protocol*[9]), we are able to automatize the generation of the proxies and the skeletons (a critical step in the hardware design flow). The final implementation is highly optimized to fit the low-cost requirements while keeping the process automatic to save design time.

4. Middleware Services and advanced features

In this section, we briefly describe those services and advanced features that differentiate OOPAmI from other middlewares. The AmI service developer point of view drives the middleware design process. We use the desirable characteristics that a global service development process should have as the guidelines for middleware design (with independence of software engineering methodology applied).

4.1 Information model

Firstly, the developer needs to know what services from the middleware can use in order to retrieve the information is going to be consumed by a new service. In the same way, only in the case the new service generates information, the developer has to know how this information must be generated (e.g. the format of the data and how the data is exported to the rest of services). Also the developer should know the interface to access other services and their attributes (including, valid values and format for these attributes).
Our information model includes the items listed above It is basically a taxonomy of services (specialization of generic services) that includes attributes for each service. This taxonomy is like our yellow pages of services and permits developers to lookup for service classes and its attributes. Each service instance running in an AmI environment belongs to a class expressed in the taxonomy and instantiates the attributes of this class. We consider this taxonomy as a mean to establish a common nomenclature for AmI service developers (aiming the same goal POSIX interfaces play in operating systems).
If we want to develop a new service (for example, modelling a new type of camera, light, virtual glasses, etc.), we should look up in the taxonomy, find the class that better fits such new service and follow the nomenclature throughout the rest of the service development process.
Ontologies are used to express the taxonomy because it is a powerful tool to visually organize information and the existent relations between the entities in the information model. Along with our ontology we define a core of basic interfaces that are used to model any service in the ontology. By aggregation, we are able to generate more complex interfaces that offer read and write operations for each attribute. The Interface Definition Language (IDL) allows us to express interfaces in a clear language (from the developers point of view) avoiding the complexity of other languages (for example WSDL for web services).
Some services need more advanced interfaces than read/write primitives. In many cases, they follow standards adopted by the industry as in the case of AVStreams from *Object*

[9] This also includes the physical interface of the hardware module (Jesús B. et al. IPSOC 2006).

Management Group (OMG, 2000a) for multimedia stream configuration, Mobile Location Protocol (Open Mobile Alliance, 2001) for spatial coordinate information and property services (OMG,2000b) for attribute management (used when low cost devices has to delegate its management to more powerful devices).
A set of basic events is also specified in the information model. With the classification of the events that can be generated in an AmI environment, the service developers have a common reference that can be use to orchestrate common behaviours. Of course, not all services will consume or generate all events. The purpose of this definition is to let the designer know a list of common events that a service could receive or generate.
The next step is to know how a service can be accessed from other services. Since we use an object model, each service has a reference which is actually a pointer to the object that implements such service. This mechanism does not differ from traditional object oriented languages but the need to get the service reference before use it.

4.2 Abstract Service Discovery Protocol (ASDF)

To allow developers to find any reference to any service in the AmI environment (note that the services are deployed in a highly distributed environment) we have defined an interface for service announcement and discovery. This ASDF interface is the result of thorough study of the service discovery protocols (SDP) most widely used in the industry and looks as follows:

```
module ASD {
        dictionary<string , Object> PropDict;
        interface iListener {
                void adv(Object_ prx , iProperties : : R_ prop) ;
                void bye(Ice : : Identity oid) ;
        } ;
        interface iSearch {
                void lookup(Object_ cb , PropDict query) ;
        } ;
} ;
```

As the reader can see the interface is clear and simple to understand (the same interface in WSDL would take almost one page in XML with multiple references).
There are two interfaces that conforms the ASDF definition: *iListener* for announcements and *iSearch* to lookup new devices supporting specific services. It is not mandatory for a device to implement the two interfaces (for example a basic sensor probably only sends announces to the environment and it does not need to lookup for nothing).
Those services which send an advertisement about its presence (invoking the *adv* operation) attach to this invocation their references (*prx*) and a reference to the service that manages its properties. In the case of services conceived to run in powerful devices, the interface *iProperties* can be instantiated. The properties interface defines the capability of a device to manage its own attributes. If this interface is not present (i.e. in the case of less powerful devices) such responsibility is delegated to other object in the system so a reference to it must be provided (the process is transparent to the clients). The later scenario is very common since it is highly flexible.
When a service invokes an adv operation, it propagates an event to a well known event channel (labelled as "ASDA") where other services in the AmI environment have previously

subscribed. The management of this event channel is provided by the middleware and can be configured with replication (for fault tolerance), with QoS parameters and federated associations for scalability purposes.
In the case of a lookup operation, the service interested in finding other services creates a temporal event channel. Then, the services that fit the properties included in the *query* parameter send an advertisement. We offer the option of a repository of services that receives all adv announcements and answers to lookup operations.
With ASDF, the developer does not need to configure any fixed reference to other services and only needs to lookup for the desired services. Sending advertisements to well event channels is the mechanism implemented to offer the functionality of a service to the rest of environment.

4.3 AmI simulator

Testing non trivial AmI services can be difficult due to the heterogeneity of possible AmI environments and situations that can take place in this environment. One of our actual ongoing works is the creation of an integrated simulation environment including:

- A set of dummy services that implement real interfaces.
- An AmI Specification Language (AmISL) to model:
  - The physical environment with the device placement.
  - Human entities and their behaviour inside the virtual environment (movements, actions and interactions activities).
  - The state of the environment (temperature, light, state of doors, etc.)
- A simulation engine that interprets the actions expressed in AmISL. For example, the synthetic data generation simulating sensor activity.
- A log tool that records the information of the service that is being tested, focusing on its interaction with the dummies services.

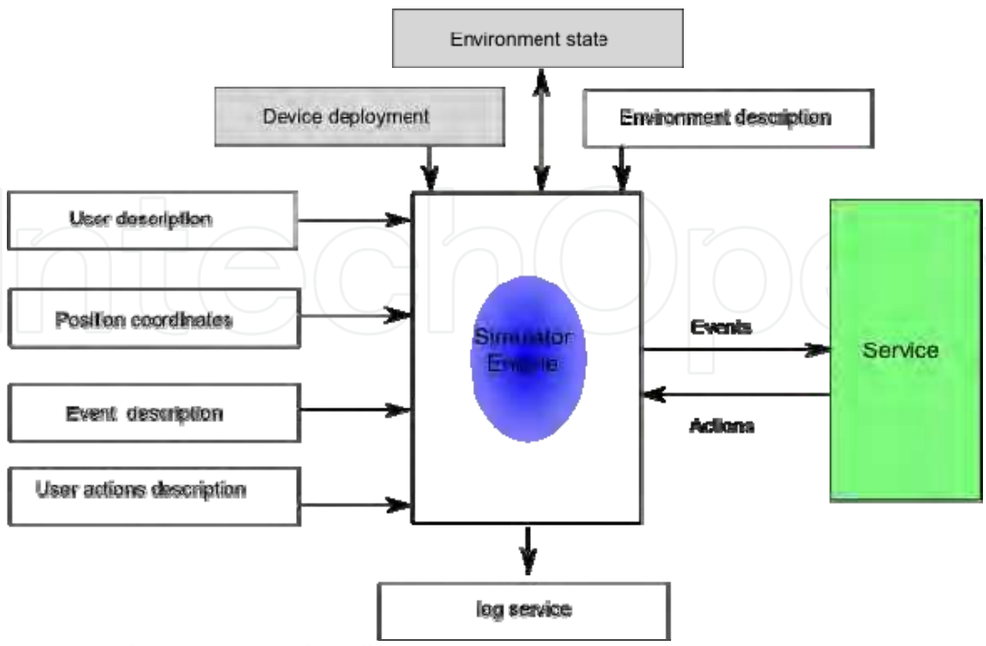


Fig. 5. AmI simulation engine data flow

In figure 5 we can see the information flow involved in the simulation process.
Once the service has been simulated and tested, the developer needs to deal with problems like *how do I deploy/stop/play/actualize my service* in the environment or *what do I have to do to obtain the initial references to the basic middleware services* (i.e. event channel service).

4.4 BootStrap service

The *bootstrap* service has been developed for easy integration of services and devices. When a device is begin deployed in any environment, it is necessary to do some tasks before offering the functionality to the rest of services in the AmI environment. With independence of the procedure used by the device to get network connectivity, the bootstrap service starts automatically a broadcast query looking for an entity called *environment manager*, this environment manager maintains the references to basic services. Each device runs a *service manager*, a directory of the services that run in this device.
The environment manager runs in a node labelled as the *coordinator*. The bootstrap service is a distributed algorithm for choosing the *coordinator* and its *replicas*. Each candidate replica can promote to the *coordinator* role when the coordinator fails.
When the *bootstrap* service of a new device starts to run, the *bootstrap* service tries to identify the coordinator of the environment and gets the reference to the environment manager. With this reference the *service manager* is then set up, the references to the basic middleware services are retrieved and the local services start to run.
As in the ASDF service, the *bootstrap* service tries to minimize configurations procedures for service and device integration using a Place&Play philosophy, similar to the successfully Plug&Play followed by the hardware industry.

4.5 Transparent dynamic reconfiguration

The reconfiguration service in our middleware has the capability of instantiating new hardware services in a transparent manner (Rincón et al., 2009).
A reconfiguration procedure implies the modification of the physical interface of the module to be reconfigured, so we need isolate the reconfigured area. Following the distributed object oriented paradigm, the hardware version of proxies and skeletons enable us to have a fixed interface (the one with the bus or network interface) so we can instantiate into any reconfigurable area.
Other problem related with reconfiguration is state persistence. Due that we know the variables that have all information related with the state of the object and its size (at design time) we know the information that we have to save and restore.
We extend the hardware adapters that translates bus read and write operations into object invocations with the following operations: *stop*, *start*, *getState*, *setState* and *initState* which are used for start, stop, get and set the state and reset a hardware object respectively.
Inside of a hardware node with reconfiguration capability (for example, an FPGA) we define a set of entities and services for a transparent reconfiguration procedure:

- *Memory Allocation Service*: this service is a centralized memory management entity for the whole system that presents a well known interface, completely independent from a concrete implementation technology or memory hierarchy.
- *Object Location Service*: the location service contains a table of references where hardware object identities are linked with valid endpoints.

- *Object Factory*: the factory service physically instantiates an object into a reconfigurable area. We can create objects of any type at run time providing its class type and a reference to the memory location of the partial bitstream (the binary image of the code).
- *Reconfiguration controller*: this entity has the responsibility of run-time creation and destruction, including object state persistence management. This service is built upon the three basic ones described before (memory allocation, location and factory).

Each of these services are objects themselves. This means that they can be implemented as HwOs or software entities. Even due to the external communication capabilities of a hardware node the reconfiguration process can be initiated and monitored from outside the chip. A possible organization will leave the reconfiguration controller and location service outside the hardware node whereas the factory and memory allocator services are attached to the hardware implementation for the best performance.

The reconfiguration process can take place explicitly or implicitly. The former is the case of an implementation of a migration service (or a application) that schedules the instantiation of HwOs and their movement across different hardware nodes. The later happens when a method of a dynamic HwO is invoked and it is not loaded in any reconfigurable area. In this case, the reconfiguration controller takes the control and instantiate the HwO without the intervention of the application that generated the invocation.

4.6 Location service

There are several position systems based in different technologies (Wifi and Bluetooth cells, RFID technology, etc.) for indoor scenarios. These systems can be combined to potentially improve the accuracy of positions of people present in the environment. Our location service combines all these systems (and the outdoor facto standard, the Global Position System) with the *Mobile Location Protocol* (Open Mobile Alliance, 2001) in order to provide a service for people location and identification. We define a set of interfaces to add new positioning systems and a set of rules to combine the information coming from such systems.

For example, the Bluetooth interface present in mobile phones can give us a clue about the area where the owner of the mobile phone is present. We can also combine the Bluetooth information with movement sensors, spread in the environment, in order to improve the accuracy of our location service. The defined interfaces enable us to use events from the environment as triggers to notify variations in the scene. For example when a door is opened or closed, a desktop session in a computer begins, the turning on or turning off of devices, detection of faces, etc.

In the location service design, we emphasize the modularity of the system in such a way that we use plug-ins to provided new functionality and “inference” rules about location coordinates.

4.7 Service Composition

Providing service composition in an autonomous manner is not trivial, and it is not achieved yet without involving users, at some level. The approach integrated in OOPAmI proposes a multidisciplinary approach, in the shape of a layered architecture. Founded on the aforementioned middleware platform, a multi-agent system is deployed on top of it. These intelligent agents retrieve information from the context, in order to provide the reasoning engine with the context information required to provide an adequate response to the current situation. This response will be provided in terms of the basic services, that when combined in a plan, will outcome the composite service.

The combination of these different technologies is successfully achieved by counting on a unique semantic model, implemented by the middleware framework, the multi-agent system and the reasoning engine. This semantic model, or ontology for a service-oriented architecture, basically considers the following entities: device, service, action, object, and property. *Devices* provide *services*, and are described in terms of *properties*, such as their location, or provided features. *Services* are described in terms of *actions* performed over *objects*, and also hold *properties.*

An OWL description of this semantic model is translated into ICE interfaces, so that all services provide a common access method. The same OWL description is used by the multi-agent system, not only for message exchange, but for interacting with the middleware services. Finally, the reasoning engine resorts to the same semantic model to describe the domain knowledge for the deployed context. Therefore, the results of the inference and search processes can be carried out by the multi-agent system.

5. Conclusions

In this chapter we have shown our middleware guidelines for a holistic approach to AmI environment development. The object oriented paradigm drives the development of the middleware services making it easier their utilization and modelling. Besides, the integration of heterogeneous devices is performed in a seamless way.

The completeness of our approach includes service development support, service discovery facilities, a tool-chain for the most complicate tasks in the case of HwOs and embedded software generation, a simulation framework and more.

Current work is focused on the extension of the pool of services that can be offered to the industry and service developer's community in order to increase the capabilities of the middleware.

6. References

Barba J.; Rincón, F.; Moya, F.; Dondo J.; Villanueva, F.J.; Villa D. & López J.C, (2007). OOCE: Object-Oriented Communication Engine for SoC Design. *Proccedings of 10th EUROMICRO Conference on digital System Design*, pages 296-302, ISBN: 978-0-7695-2978-3, August 2008.

Friedewald, M. & Da Costa, O., (2003). Science and Technology Roadmapping: Ambient Intelligence in Everyday Life, technical report, JRC/IPTS European Science and Technology Observatory. Available online at:: http://www.cybertherapy.info/pages/AmIReportFinal.pdf (last visit Sept. 15th 2009)

Gelissen, J. (205). IST-Ozone Project Final Report, technical report, Available at : http://www.hitech-projects.com/euprojects/ozone/public_docs/ozone-phr-19Jan05-final-report-jg.pdf (last visit Sept. . 15th 2009).

Henning, M. & Spruiell, M., (2008). Distributed Programming with ICE, version 3.3. At http://www.zeroc.com (last visit Sept. 15th 2009).

Issarny, V. ; Sacchetti, D. ; Tartanoglu, F. ; Sailhan, F. ; Chibout, R. ; Levy, N. & A. Talamona (2005). Developing Ambient Intelligence Systems: A Solution based on Web Services. *Journal of Automated Software Engineering*, Vol. 12, Issue 1, pages 101-137, January 2005, ISSN : 0928-8910.

IST Advisory Group (2003). *Ambient Intelligence: from vision to reality*, technical report. At: http://www.cordis.lu/ist/istag-reports.htm (last visit Sept. 15th 2009).

Maly, I. ; Curin, J. ; Kleindienst, J. & P. Slavik. Creation and Visualization of User Behavior in Ambient Intelligent Environment. *Proccedings of 12th International Conference Information Visualization*, pages 497-502, ISBN: 978-0-7695-3268-4, July 2008, IEEE Computer Socity, Whashington DC.

Marsa, I. ; López-Carmona M.A & Velasco, J. R. (2007). A hierarchical, agent based service oriented architecture for smart environments *in Service Oriented Computing and Applications*, Springer, ISSN : 1863-2386, London.

Object Management Group (2000). *Audio/Video Stream Service*. OMG Document formal/00-01-03.pdf, 2000a.

Object Management Group (2000). *Property Service Specification*. OMG Document formal/00-06-22.pdf, 2000b.

Open Mobile Alliance (2001). *Mobile Location Protocol Specification*. Document Location Interoperability Forum TS 101 Specification, version 3.0.0. Available online at: http://www.openmobilealliance.org/tech/affiliates/lif/lifindex.html (last visit Sept. 15th 2009).

OSGi Alliance (2006). *OSGi Service Platform: Core Specification*, edition 4.0.1 Release 4, July 2006. At : http://www.osgi.org/Release4/Download (last visit Sept. 15th 2009).

Perumal, T. ; Ramli, A.R. & Yew Leong, C. (2008). Design and Implementation of SOAP-Based Residential Management for Smart Home Systems. *IEEE Transactions on Consumer Electronics*, Vol. 54, Issue 2, pages 453-459, ISSN : 0098-3063.

Ramos, C ; Augusto, J.C. & D. Shapiro (2008). Ambient Intelligence- the next step for Artificial Intelligence. *IEEE Intelligent Systems*, Vol. 23, Issue 2, pages 15-18, March-April 2008, ISSN : 1541-1672.

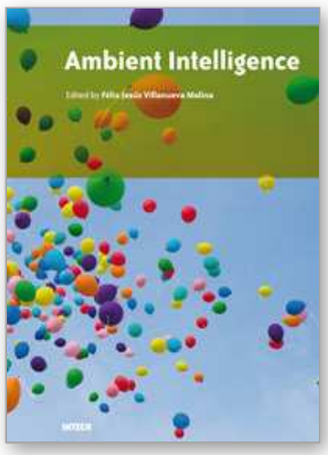
Rincón, F.; Barba J.; Moya, F.; López J.C & Dondo J. (2009). Transparent Dynamic Reconfiguration as a Service of a System-Level Middleware, *in Lecture Notes in Computer Science*, Springer, ISSN: 0302-9743, Berlin.

Sacchetti, D. ; Bromberg, Y.-D. ; Georgantas, N. ; Issarny, V. ; Parra, J. & R. Poortinga (2005). The Amigo Interoperable Middleware for the Networked Home Environment. At : http://middleware05.objectweb.org/WSProceedings/demos/d4_Parra.pdf (last visit Sept. 15th 2009).

Villanueva, F.J.; Moya, F.; Santofimia M.J.; Rincón, F.; Villa D.; Barba J. & López J.C, (2009). Towards a Unified Middleware for Ubiquitous nad Pervasive Computing, I*nernational Journal of Ambient Computing and Intelligence*, Vol. 1, Issue 1, pages 53-63, ISSN: 1941-8647.

Wu C. & Fu, L. (2008). A Human-System Interaction Framework and Algorithm for UbiComp-Based Smart Home. *Proceeding of 2008 Conference on Human System Interaction*, pages 257-262, ISBN: 978-1-4244-1542-7, May 2008.

IntechOpen



Ambient Intelligence

Edited by Felix Jesus Villanueva Molina

ISBN 978-953-307-078-0
Hard cover, 144 pages
Publisher InTech
Published online 01, March, 2010
Published in print edition March, 2010

It can no longer be ignored that Ambient Intelligence concepts are moving away from research labs demonstrators into our daily lives in a slow but continuous manner. However, we are still far from concluding that our living spaces are intelligent and are enhancing our living style. Ambient Intelligence has attracted much attention from multidisciplinary research areas and there are still open issues in most of them. In this book a selection of unsolved problems which are considered key for ambient intelligence to become a reality, is analyzed and studied in depth. Hopefully this book will provide the reader with a good idea about the current research lines in ambient intelligence, a good overview of existing works and identify potential solutions for each one of these problems.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen