Case Study of the OMiSCID Middleware: Wizard of Oz Experiment in Smart Environments

Rémi Barraquand, Dominique Vaufreydaz, Rémi Emonet and Jean-Pascal Mercier
PRIMA Team - INRIA
Zirst 655, avenue de l'Europe
38334 Saint Ismier cedex
{*remi.barraquand,dominique.vaufreydaz,remi.emonet,jean-pascal.mercier,omiscid-info*}*@inrialpes.fr*

***Abstract*—This paper presents a case study of the usage of OMiSCID 2.0, the new version of a lightweight middleware for ubiquitous computing and ambient intelligence. The objective of this middleware is to bring Service Oriented Architectures to all developers. After comparing to available solutions, we show how it integrates in classical workflow without adding any constraint on the development process. Developers only need to use a library available in widely used programming languages (C++, Java and Python). Then, the basics of OMiSCID and a brief technical description are described as its *User Friendly API*. This API makes it straightforward to expose, look for or send messages between software components over a network. The added value of the proposed middleware has already been experienced in international research projects. This paper demonstrates its effect in cutting down development time, improving software reuse and easing redeployment in the context of a Wizard of Oz experiments in intelligent environments.**

***Keywords*-Service Oriented Architecture, Ubiquitous Computing, Middleware, Wizard Of Oz, Smart Environments.**

I. Introduction

Today's vision of ubiquitous computing is only half way achieved. The multiplication of low cost devices along with the miniaturization of high performance computing units technically allow the design of environment widespread by cameras, motion detectors, automatic light controls, pressure sensors or microphones. Those devices, given the apparition of wireless networks, can communicate together with mobile and personal equipments such as cellular phones, photo frames or even personal assistants. On the other hand, the quiet and peaceful aspect of this vision where computing units can understand each others in order to collaborate is yet a research problem.

Build upon this network of devices, ambient intelligence tries to address the problem of making devices refer to user in an appropriate way by making them aware of his activity: current task, availability, focus of attention, etc. Such environments that sense user activity and act according to it are named "intelligent environments" or "smart environments".

However, activity understanding is yet a complex problem and relies on the ability to constantly aggregate information from an ever changing medium where devices come and go, break and evolve. Often used only to refer the access to information or applications from occasionally-connected devices, mobility is a key aspect of this vision.

For the user experience to be optimal anywhere anytime, ambient intelligence should be mobile. The intelligence can for example be embedded in a cell phone and dynamically adapt to the current environment and what services it provides. Ambient intelligence covers numerous fields and poses many challenges. Among these, handling dynamicity in software architecture is one that plays a central role.

This paper addresses the use of the OMiSCID [1] middleware that fits between the network of devices and ambient intelligence. It aims to ease the design of agile Service Oriented Architecture (SOA) and to solve constraints of pervasive computing and intelligent environments. OMiSCID orchestrates services in the environment, by providing cross-platform/cross-language tools for easy description, discovery and communication between software components.

The next section introduces the needs for such middleware. In section III, we present our approach focusing on key functionalities and concepts. Usage of OMiSCID is then illustrated by the design of a Wizard of Oz experiment. Finally, we draw some conclusions.

II. Ubiquitous Computing Requirements

The PRIMA team works on perception and perceptive spaces (using multiple cameras, microphones, wireless technologies, etc.), context awareness and ambient intelligence. In such research area, many services, developed by specialists using multiple techniques and languages, must interconnect in order to achieve a final goal. In the context of international researches, like DARPA or EU project funding, this problem is even more important because of different habits and of historical technical backgrounds within the involved research groups. In order to help non software-architect researchers to interact, we need a simple and usable solution that addresses a common problem: how to find, interconnect and monitor services within the context of cross-language cross-platform distributed applications?

To solve this problem, one can envision many different approaches. The first one is to decide, *a priori*, what will be the common way of programming. This solution can be adopted in small groups and must be driven by underlying technologies. It may make people learn a new language or framework to interoperate and can changed every time involved scientists are replaced by new ones. The second possible scenario is to list all the software pieces and try to find a common way to do networking. In ubiquitous computing and ambient intelligence, this list will exhibit a variety of criterions:

- programming languages: C++ for video/audio processing, Java for lighter processing or user interfaces, scripting language.
- operating systems: Windows (for device support for instance), Linux, Mac OSX (cross-platform);
- data to exchange: simple texts, data structures or huge data like audio/video flows (messages or stream flows);
- configuration: peer to peer connection (IP address/port) or more complex interconnection scripts using service description and recruitment (dynamic service discovery);
- maintainability and sustainability.

Their are three main available solutions, the two most widely used are compared in the Table I. OSGi [2] is the first typical solutions to provide most of the requirements listed formerly. It permits construction of Java applications locally by recruiting components. Using iPOJO [3], it is possible to describe services and requirements in order to avoid writing dedicated code. Using specific adapters, like in R-OSGi [4] it is also possible to search for non local services. H-OMEGA [5] proposes also an alternative using UPnP for device discovery and a centralized server for code management. Nevertheless, even if it is always possible to use JNI for C/C++ application, OSGi is dedicated to Java. We do not retain this solution.

Web Services [6] are also a widely used solution for distributed applications. They permit using web technologies to construct distributed applications. Services are described with the *Web Services Description Language* (WSDL) and can be discovered using *WS-Discovery*. As said in Table I, *Web Services* are not designed for huge stream flows. Moreover, event if there are several alternatives to WSDL, like BPEL4WS[1] [7] or OWL-S[2] [8], they all provide service descriptions that are not easy to handle for a non specialist. We do not retain this solution.

The last possible solution is to use one of the specialized middlewares usually dedicated to a specific task and/or environment. We can illustrate them by focusing on *Smart Flow II* [9]. This middleware is very efficient in managing the data flow from many multimedia sources at the same time on several computers. But its force is also its weakness: it is difficult to configure and to manage other type of data.

From the previous sections, we can see that none of these solutions fulfills all the identified requirements. This assumption was the start of the OMiSCID middleware development. In the following sections, we will present the underlying concept and philosophy behind OMiSCID middleware solution.

[1]*Business Process Execution Language for Web Services*
[2]*Web Ontology Language for Services.*

III. OMiSCID Basics

OMiSCID stands for Opensource Middleware for Service Communication Inspection and Discovery [1]. It was designed and built to answer the problem of integration and capitalization of heterogeneous code inside augmented environment. OMiSCID is distributed with a non sticky MIT-like license, fully open source and available on a dedicated website: http://omiscid.gforge.inria.fr/.

A. Concepts

The OMiSCID middleware is built around 3 main concepts: services, connectors and variables. A service is a piece of software with a well defined way to access its provided functionalities. Services are composed of connectors and variables. At least, a service has the following read only variables: a name, a class, a unique service id and login/hostname information. Connectors are used to transfer data between services and are either input, output or both. Variables describe the service or its state. They may have local or remote write protection. Aggregating all these information, we obtain a service description that can be used to search and interconnect services.

B. Communications

Messages are atomic elements of all communications in OMiSCID. They are sent using a connector to a specific peer or to all listening services at once. The receiver will be notified that a new message is ready when it is fully available. Each message is provided with contextual information such as the service and connector it comes from. There are 2 main kinds of workflow for messages that can be mixed:

- A peer to peer approach. After receiving a message from a service and processing it, a response message is sent back to it;
- A data flow approach. After receiving a message on a connector and processing it, a message with the result is broadcasted on another connector in order to continue the processing chain.

Message can be sent as raw binary chunk or as text, which allows lot of flexibility for developers. Binary messages are often used to stream real time data such as video or audio. Text communication can be enhanced by the JSON format and allows for more advanced operation and extensions (see Section III-D).

C. Service discovery in dynamic context

Also known as service discovery, the ability to browse, find and dynamically bind running services, is one of the most important features of SOA moreover in ubiquitous

Table I
COMPARISON OF WIDELY USED SOLUTIONS

Description	cross-language	cross-platform	Messages	Huge data flows	Service discovery over the network
OSGI approach	No (Java)	Yes	Yes	Possible	Using *R-OSGi* for instance
Web Services	Yes	Yes	Yes	Not designed for	Using *WS-Discovery*

environments. It is common to filter services based on their current state, description or functionalities. OMiSCID provides the basic logical combination of predefined search criteria (variable value/name, connector properties, etc.) and since they are implemented as functor (function object), it is easy to extend it with user defined filters. Filters can be used in two different ways:

- An ask-and-wait approach asking for the list of services that match a certain criterion. This procedure will wait until at least one service match or that a timeout is reached rising an exception.
- An ask-and-listen approach notifying the application by the means of callback or listener whenever a service that matches the criterions appears or disappears.

Figure 2 and Section V-C gives clues about OMiSCID service discovery capabilities.

D. Serialization and remote procedure call

The philosophy of OMiSCID is to exchange information between services using either binary or textual messages without any standard on the format of those messages. OMiSCID 2.0 provides a simple way to marshal and unmarshal any object. This allow for easy communication between services without paying attention to serialization issues.

An OMiSCID service can expose some of its functionalities by the means of remote callable methods. Such distant calls can be done either in a synchronous or in an asynchronous manner.

E. OMiSCID Gui

OMiSCID provides a simple solution to declare, to discover and to interconnect services. Nevertheless, it needs, for maintainability and sustainability, an interface to visualize and to debug distributed services. OMiSCID Gui is a powerful tool built over the Netbeans platform and provides the developer with a graphical interface for multiple management tasks. It inherits many of the advantages from the Netbeans platform: portability, modularity, advanced window management, etc... OMiSCID Gui comes with light core modules and is extensible at infinite (see [10] for details). One of the core modules is a service browser that displays all the services present within the environment as well as their connectors and variables. This module also provides many contextual and extensible operations to be applied on the listed services and their variables/connectors. Among all the extensions available and easily installed using the Netbeans Plugin interface, one can find:

- A simple variable plotter than can dynamically create and display graphics of remote service variables.
- A family of plugins that allow displaying 2D information such as video stream or custom shapes representing for instance regions of interest of a 3D tracker.
- A plugin that displays a graph of the services in the environment along with their interconnections.
- A lot of other plugins such as: real time audio stream player, 3D visualization tools, cameras controls etc...

OMiSCID Gui comes with a public plugin repository already packed with visualization, controls, debugging plugins and can be extended by developers. All Netbeans platform plugin can also be integrated to our platform. Its ease of use make it a must-have tool for OMiSCID development, demonstration or service oriented application development.

IV. BRIEF TECHNICAL DESCRIPTION

One crucial requirement when designing a middleware for such heterogeneous research area is to make it usable by most of the people involved.

A. Multiplatform/Cross-Language

OMiSCID was designed with cross-platform/cross-language capabilities in mind. There are actually 3 supported implementation of OMiSCID: C++, Java and Python[3] (PyMiSCID). Moreover, the Java version can be used from Matlab and any other language running on the Java virtual machine (JavaFX, scala, groovy, javascript, etc.) We also provide an OSGI abstraction layer that exposes OMiSCID with standard OSGI paradigm. OMiSCID was developed more as a set of guideline rather than a specification, all the implementations are fully written in the target language, thus ensuring speed, reliability, close integration with data structure and programming paradigm.

All versions are fully cross-platform and works on Linux, Windows and OS X both 32 bits and 64 bits. The C++ version uses an abstraction layer that provide common system objects like sockets, threads, mutexes, etc. The Java version can be used on portable device like PDA. All implementations on all supported platforms can interoperate with each other.

B. User Friendly API

In order to simplify interpersonal communications between OMiSCID users, we developed a common *User*

[3]Formerly, Python version was a simple wrapper to the C++ source code.

Friendly API. It was defined to be easy to learn, easy to use and portable in several languages. Indeed, concepts, methods, parameters follow the same API in C++, Java and Python. However each implementation takes advantage of the language specificities and design patterns.

The API also provides simple callback/listener mechanisms. Thus, one can be notified of many different events: a new connection from a service, a disconnection, a new message, a remote variable changed, etc.

C. Performance and scalability using OMiSCID

For the discovery process, registering 100 services over the networks from 3 differents computer takes less than 1 seconde. Searching for a service among hundreds using a simple variable value is less than 20 ms long. More specific searches using user-defined search filter (processing video stream to select a camera for instance) can obviously spend much more time. Latency for sending messages is around 4 ms in average comparing to usual TCP/IP connections. Detailed tests and explanations can be found in [10] and on the OMiSCID Web site.

V. Case Study

For the past few years OMiSCID middleware has been used in different research projects [11], [12], [10], [13]. In [10], OMiSCID is used to redesign a complete 3D Tracking system as well as an automatic cameraman. The redesign reduces the number of software components and has the advantage to provide shared and reusable services. For instance, both architectures use the same video grabbing services. This service provides real time streaming of camera images, that is simultaneously accessible by multiple remote services such as visualization services or image processing ones: movement detector, person detector, posture estimator. OMiSCID middleware allows robustness for service discovery, reliable communication, connection and disconnection but also ease the federation of services. In [13], [12], OMiSCID is used for the realization of a smart agent. The perception of the agent is provided by services dynamically discovered in the environment allowing the agent to construct a situation model [14] of the current situation. The agent is yet another service and, according to its perception, it is able to perform actions in the environment by sending orders to actuator services. In [13], the knowledge of the agent is distributed and can be stored on remote database using a combination of OSGi and OMiSCID. Each service developed is a reusable piece of software that ensures a decrease of development time along the years. The Wizard of Oz (WOz) experiment we conducted is the perfect example.

A. Requirements for Wizard of Oz

WOz experiment is a research experiment, in which subjects interact with a computer system (fake mobile phone, remote controlled robot, etc.) that subjects believe to be autonomous, but which is actually being operated or partially operated by an unseen human: the wizard. The goal of such experiments is to evaluate system's functionalities without actually implementing them for real. The functionalities validated by the experiments can then be implemented while reconsidering others, saving money and time. In most settings, subjects are located in a room along with the system to evaluate while the wizard operates in another room. Both rooms can be separated by a beam splitter allowing the wizard to observe and react accordingly to the subject(s) actions.

Setting up a WOz experiment requires an important preparation, even more if the settings have to be mobile and re-deployable: to be carried in different places. The wizard must have access to a multitude of information in order to control the system as best as possible. Without the presence of a beam splitter, the environment must be equipped with cameras, microphones and speakers to record and stream the scene in real time. Among those devices, the wizard also needs the proper controllers to remotely manipulate the system, which requires an extensive use of wireless or wired communication between software components: controllers and controllees. In addition, the coupling between software components has to be able to change and to be easy to achieve. Allowing for instance to deploy debuggers, loggers or visualization tools at runtime.

B. Experimental Settings

On an ongoing research project we sought to evaluate the behavior of subjects immersed in a ubiquitous environment while asked to teach a smart agent how to control the space. Among few, the objectives were to validate hypothesis about human-machine interaction as well as to collect constructive outcomes that will help future design of ambient systems. Four kinds of actor are to be considered in this experiment: the subjects, the smart agent, the environment and the wizards.

The subjects by group of 2 or 3, were asked to teach the agent how to control the environment in order to organize a small meeting. For instance a common task is to teach the agent to switch on the light when people enter the room and to switch it back off when everybody is leaving.

The agent is embodied by a personal mobile phone with wireless capabilities, on which we deployed a learning software and a simple user interface. This interface allows collecting real-time feedbacks from the subjects (good, bad) during the session. The agent connects through the wireless network to a situation modeler service that provides a situation model [12] of the current situation. When requested by the subjects the agent takes action in the environment by sending orders to actuator services. Subjects can reward the agent whenever they agree or disagree with its action. Doing so the agent learns to control the environment using dynamically discovered services and feedbacks provided by users.

The environment is an office (Figure1) spread with many actuators and sensors. OMiSCID allows each of them to be accessible and controllable by services all-over the network (Figure2). Among those sensors we find cameras, microphones, thermometer and weather station. All the actuators are controllable by OMiSCID connectors and their states can be queried by those connectors or are exposed through variables. Among the actuator we list: a steerable projector, x10 controller, speaker or even shutter. For this experiment we needed two wizards.

The first wizard was in charge of simulating certain actuators in the environment such as pressure detectors under the chairs and sofas. Indeed, we didn't dispose of such device in our environmental facility, thus we emulated those actuators using a user interface plugged into OMiSCID Gui. The simulating interface was seen as yet another service that can be used by the situation modeler to build up a better situation model.

The second wizard was controlling the overall experiment using a master interface. This interface allowed writing real-time observations through an annotator service, as well as taking control over all the services in the environment. Such a master control for instance let the wizard speed up the experiment by helping the agent to guess better actions (when subjects got exhausted), or by putting back the environment settings in an appropriate state.

C. OMiSCID At Glance

For this experiment we deployed more that 20 services spread on 5 computers running different operating systems (Linux, Windows, MacOSX). Figure 2 presents some of the devices present in the environment as well as the interconnection of services. Due to the complexity of the schema some services have been removed. In the following we review some of the advantages of using OMiSCID in this experiments:

1) Multi-platform: 5 computers have been used during the experiments, two of them by the wizards. One of the wizards was using MacOS, on which we deployed the master control. Due to driver issue the sound recording system was using a Microsoft powered computer. The video streaming as well as all the other services (archiver, x10, etc) were running on Linux hosts.

2) Multi-language: To design the services we made use of different languages. C++ was used for performance reason such as for the video and sound processing/capture services. Python is a really powerful language for the rapid prototyping of application, we used python to quickly develop the x10 controllers or the PanTilt controller. Java has been used to develop the OMiSCID Gui modules but also to access the different web services present in the environment such as the weather service. JavaFX was used to develop the wizard control's interface, its script language make it easy to use for inexpensive user interface design.

3) Service Discovery: The simple but powerful service discovery system provided by OMiSCID has been used to dynamically connect services together. The best examples are the *situation modeler* and *archiver*. Using a service repository they were able to filter services present in the environment in order to connect to them. For instance the *situation modeler* was looking for all services having connector or variable exposing state information. Using that state information, it was able to provide a situation model on an output connector. The *archiver* was responsible to backup any information transmitted between services on a hard drive. The archiver was looking for all services having output connector. Thus it was easy for instance to deploy or shutdown services on the fly during the experiment.

4) Communication: Communication between services was achieved using different format. For video and sound services, data were raw binary information tagged with time stamps. Web services such as the weather provider were communicating information using XML on their connector. The PanTilt controller exposed its commands by the means of remote callable methods, and presented its internal state using readable variable.

5) OMiSCID Gui: OMiSCID Gui was used by the wizard for different purpose. Firstly, the streamed sound and video were played by the embedded player. Indeed, we have developed OMiSCID Gui modules to play video and listen to audio stream in real-time. Those modules were used to have a feedback of what was happening into the experimental facility disposed into another building. Secondly, OMiSCID Gui was used to control the archiver and other services.

6) Reusability: Each of the service used in this experiment is a reusable piece of software that can be carried and deployed easily. For a wizard of Oz experiment only the hardware and the equipments (cameras, microphones) have to be transported and reinstalled. Everything else is deployable instantly and can adapt to the configuration: number of computers, operating systems, number and nature of devices, etc...

VI. Conclusion

OMiSCID provides a complete but simple solution to declare, describe, discover and interconnect services as well as to manage their inter-communication. OMiSCID offers to researchers several facilities for ubiquitous computing with its multi-platform and cross-language capabilities. The underlying concepts as well as the API are user friendly and directed toward usability. Along with this middleware, OMiSCID Gui provides developers with an extensible, portable and modular platform that ease development and debugging, and improve maintainability of OMiSCID demonstrations and applications.

OMiSCID has successfully been used in several academic research projects and more recently in a wizard of Oz experiment. Such an experiment requires an important amount

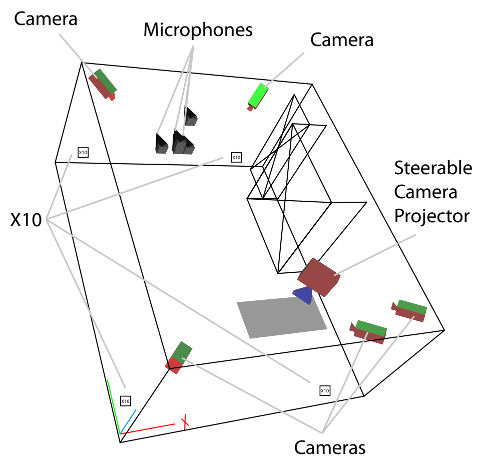

Figure 1. PRIMA's Smartroom.

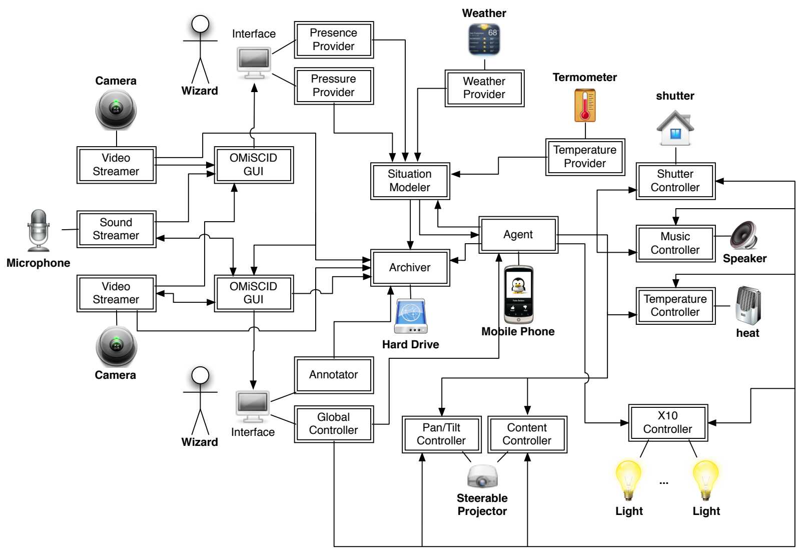

Figure 2. Wizard of Oz Services.

of resources and preparations, particulary when realized in smart environments. We have presented how the use of OMiSCID and OMiSCID Gui greatly reduces development time, maximizes reusability and eases redeployment. Furthermore, this solution does not rely on the number of computers, operating systems or network configuration.

REFERENCES

[1] R. Emonet, D. Vaufreydaz, P. Reignier, and J. Letessier, "O3miscid: an object oriented opensource middleware for service connection, introspection and discovery," in *1st IEEE International Workshop on Services Integration in Pervasive Environments*, Lyon (France), jun 2006.

[2] D. Marples and P. Kriens, "The open services gateway initiative: an introductory overview," *IEEE Communications Magazine*, vol. 39, no. 12, pp. 110–114, Dec. 2001.

[3] C. Escoffier, R. S. Hall, and P. Lalanda, "ipojo: an extensible service-oriented component framework," *Services Computing, IEEE International Conference on*, vol. 0, pp. 474–481, 2007.

[4] D. Wang, L. Huang, J. Wu, and X. Xu, "Dynamic software upgrading for distributed system based on r-osgi," in *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 227–231.

[5] C. Escoffier, J. Bardin, J. Bourcier, and P. Lalanda, "Developing User-Centric Applications with H-Omega," in *Mobile Wireless Middleware, Operating Systems, and Applications - Workshops*. Springer Berlin Heidelberg, April 2009, pp. 118–123.

[6] M. Papazoglou, *Web Services: Principles and Technology*. Prentice Hall, September 2007.

[7] R. Khalaf, N. Mukhi, and S. Weerawarana, "Service-oriented composition in bpel4ws." in *WWW (Alternate Paper Tracks)*, 2003.

[8] D. Martin, M. Paolucci, S. Mcilraith, M. Burstein, D. Mcdermott, D. Mcguinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, "Bringing semantics to web services: The owl-s approach," in *SWSWPC 2004*, ser. LNCS, J. Cardoso and A. Sheth, Eds., vol. 3387. Springer, 2004, pp. 26–42.

[9] A. Fillinger, L. Diduch, I. Hamchi, M. Hoarau, S. Degre, and V. Stanford, "The nist data flow system ii: A standardized interface for distributed multimedia applications," in *World of Wireless, Mobile and Multimedia Networks, 2008. WoWMoM 2008. 2008 International Symposium on a*, 23-26 2008, pp. 1 –3.

[10] R. Emonet, "Semantic description of services and service factories for ambient intelligence," Ph.D. dissertation, Grenoble INP, sep 2009.

[11] J. L. Crowley, D. Hall, and R. Emonet, "Autonomic computer vision systems," in *Advanced Concepts for Intelligent Vision Systems, ICIVS 2007*, J. Blanc-Talon, Ed. IEEE, Eurasip,, Aug 2007.

[12] R. Barraquand and J. L. Crowley, "Learning polite behavior with situation models," in *HRI '08: Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*. New York, NY, USA: ACM, 2008, pp. 209–216.

[13] S. Zaidenberg, P. Reignier, and J. L. Crowley, "An architecture for ubiquitous applications," *Ubiquitous Computing and Communication Journal (UBiCC)*, vol. 4, no. 2, jan 2009.

[14] J. L. Crowley, P. Reignier, and R. Barraquand, "Situation models: A tool for observing and understanding activity," in *in Workshop People Detection and Tracking, held in IEEE International Conference on Robotics and Automation*, Kobe, Japan, 2009.