The OMiSCID 2.0 Middleware: Usage and Experiments in Smart Environments

Rémi Barraquand*, Dominique Vaufreydaz†*, Rémi Emonet*, Amaury Nègre*, Patrick Reignier‡*
*PRIMA Team - INRIA/LIG/CNRS - 655, avenue de l'Europe - 38334 Saint Ismier Cedex
see http://www-prima.inrialpes.fr/
†Université Pierre-Mendès-France - BP 47 - 38040 Grenoble Cedex 9
‡Université Joseph Fourier - BP 53 - 38041 Grenoble Cedex 9
{Remi.Barraquand,Dominique.Vaufreydaz,Remi.Emonet,Amaury.Negre,Patrick.Reignier,omiscid-info}@inria.fr

***Abstract*—OMiSCID 2.0 is a lightweight middleware for ubiquitous computing and ambient intelligence. Its main objective is to bring Service Oriented Architectures to all developers. After reviewing related works, we demonstrate how OMiSCID 2.0, compared to other available solutions, integrates easily in classical workflows without adding any constraints on the development process. A basic overview of our middleware is given along with brief technical descriptions demonstrating its *User Friendly Application Programming Interface*. This application programming interface makes it straightforward to expose, look for or send information between software components over the network. We illustrate the usage of OMiSCID 2.0, the new version of our lightweight middleware, through case-studies that have been experienced in international research projects. Particularly, we demonstrate its advantages in both development and research projects, illustrating its radical cut down effect in development time, improving software reuse and easing redeployment notably in the context of Wizard of Oz experiments conducted in smart 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, thanks to 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 users in an appropriate way by making them aware of their activity: current task, availability, focus of attention, etc. In this context, *smart environments* or *intelligent environments* refer to environments that are spread with sensors and actuators which sense users' activities and respond according to them.

In this attempt, activity understanding remains a complex and challenging problem relying on the ability to constantly aggregate information from an ever changing medium of devices and media. A medium of information, which is in constant evolution due to the fact that media and devices, come and go, break and evolve. Mobility in this context is no longer an option. In order to guaranty the best user experience, services provided to the user should be available everywhere and at any-time. The *intelligent part* –the one that guarantees the best mapping between perceptions and actions, must, in some way, be carried along with the user in its daily activities and could, for instance, find itself embedded in a mobile phone. While the user will carry his mobile phone, the later will have to dynamically adapt to the current environment, connecting to available sensors and actuators, scanning for and exchanging with available services.

Ambient intelligence, thus, relies on a large number of different fields of expertise and brings along many challenges. Among these challenges, one that plays a central role is the handling of dynamicity in software architectures. This paper addresses the use of the OMiSCID [2] middleware that fits in 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 manages services in the environment, by providing cross-platform/cross-language tools for easy description, discovery and communication between software components.

In the next section, we introduce the needs for such a middleware and we present our approach, focusing on key functionalities and concepts. Benefits of using OMiSCID are shown with some *refactoring and reusability* examples. Finally, the use of OMiSCID is illustrated by the design of a Wizard of Oz experiment.

II. UBIQUITOUS COMPUTING REQUIREMENTS

The overall goal of the PRIMA research group is the elaboration of a scientific foundation for interactive environments. An interactive environment requires the capabilities of perception, action and communication. An

environment is said to be perceptive if it is capable of maintaining a model of its occupants and their activities. Such a model may include the identity of individuals, an estimation of their position, their recent trajectory, as well as recognition of the activities of individuals and groups. An environment becomes active when it is capable of action. Actions may include presentation of information. They may also include the capability to manage visual and acoustic communications, as well as the capability to transport documents and material. Controlling an environment that is perceptive and active requires a capability to interact. This capability may rely on speech recognition, gesture or object manipulation interpretation, observation of people interactions. Among those challenges is the one of developing and integrating these three capabilities.

For instance, in order to build ambient intelligence applications, many software services, developed by specialists using multiple techniques and languages, must dynamically interconnect to furnish users with the best comfort as possible. In the context of international research projects, such as DARPA or EU funded projects, the problem get even more complex as the differences among research groups involved are important. Differences include habits and historical/technical backgrounds. In order to help non software-architect researchers to interact and better collaborate, we need a simple and usable solution that addresses a common problem: how to find, to interconnect and to monitor services within the context of cross-language, cross-platform and distributed applications?

To solve this problem, one can envision many different approaches. The first one is to agree, *a priori*, about a specific programming convention, e.g., languages, platforms, technologies and so on. This solution can be adopted in small groups and must be driven by underlying technologies. This, however, has many drawbacks. For instance, it may oblige people to learn a new language or a new framework. Additionally, the agreement achieved between scientists and developer involved is subject to change, depending on the evolution of technologies but also on the evolution of the team members. An alternative scenario is to list all the software components and try to find a common way to interconnect them, no matter the platforms/languages used or any other constraints that might appear.

The alternative scenario is often approached by the implementation of a middleware, aiming to abstract lower software components. Bellow we list properties that such a middleware should have, at least in ubiquitous computing and ambient intelligence, our research area:

- *Attractiveness*: To be attractive, a middleware must be available in several programming languages (C++ for video/audio processing, Java for lighter processing, enterprise integration or user interfaces development, scripting language for rapid prototyping) on multiple operating systems (Windows, Linux, MacOSX/iOS, Android).
- *Extensibility*: To be accepted, the integration of a middleware in existing programs must be timeless, costless and effortless. Adding new functionalities must be as simple as possible.
- *Networking*: Networking capabilities must support, a various, always stretching, range of protocols like peer to peer connections (IP address/port) or more complex interconnection protocols using service description and discovery for instance. Ad-hoc networks should also be supported. The middleware must also support the exchange of various data types between the software components, from simple text message or formatted data structure to huge data like audio/video flows.
- *maintainability* and *sustainability*. Maintainability includes readability of the source code, a friendly and user oriented API, predictability of the software behaviors and monitoring of running processes over the network. Sustainability is the potential for the long-term maintenance and reuse of software components. Sustainability is strongly correlated to maintainability.

Two widely used solutions are OSGi and Web Services. They are compared in Table I. OSGi [3] 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 [4], it is possible to declaratively describe services and requirements using annotations for instance in order to avoid writing dedicated code. Using specific adapters, like in R-OSGi [5] it is also possible to search for non local services. Exposing an OSGi service to multiple protocols (like UPnP [6], Web Services, etc.) can be easily achieved using the ROSE (also named Chameleon) [7] ecosystem over an OSGi platform. H-OMEGA [8] proposes also an alternative using UPnP for device discovery and a centralized server for code management. Nevertheless, it is difficult to combine all the evolutions of OSGi. Additionally, even if it is possible to use JNI for C/C++ application, OSGi is dedicated to Java.

Web Services [9] are also a widely used solution for distributed applications. They permit to use web technologies in order to construct distributed applications. Services are described with the *Web Services Description Language* (WSDL) and can be discovered using *WS-Discovery*. As presented in Table I, *Web Services* are not designed to handle huge data flows. Moreover, even if there are several alternatives to WSDL, like the Business Process Execution Language for Web Services (BPEL4WS) [10] or the Web Ontology Language for Services (OWL-S) [11], they all provide service descriptions that are not easy to handle for a non specialist.

Finally, the last possible solution is to use a specialized middleware, usually dedicated to a specific task and/or environment. We can illustrate this solution by focusing on

Smart Flow II [12]. 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 the reviewed solutions fulfils all the identified requirements. This assumption motivated our effort to develop the OMiSCID middleware. In the following sections, we will present the underlying concept and philosophy behind the OMiSCID middleware solution.

III. OMiSCID Basics

OMiSCID stands for Opensource Middleware for Service Communication Inspection and Discovery [2]. It was designed to answer the problem of integration and capitalization of heterogeneous code inside smart environments. OMiSCID is distributed under a MIT-like license (free for both commercial and non-commercial applications), fully open source and available on our forge [13].

A. Concepts

The OMiSCID middleware is built around 3 main concepts: services, connectors and variables. They are detailed in the following sections.

1) Services: A service is a piece of software that exposes, in a transparent and light way, functionalities for a specific task. Functionalities are thus visible and available for any other services over the network without any implementation constraint. A service exports its functionalities and its state through its connectors and variables. At least, a service contains:

- *name*. This variable must represent the main function of the service. It should be human readable like *Camera*;
- *class*. This variable allows to logically organize services in categories, for instance *VideoProcessing*;
- *id*. This unique id over the network is automatically generated by OMiSCID. Services can be distinguished using this id.
- *hostname*. Computer name where the service is running;
- *owner*. *Owner* is the login which starts the service on *hostname*;
- *control port*. The *control port* is a connector used to control and manage the service.

Aggregating all these information, we obtain a service description that can be used to search and interconnect services. Services in OMiSCID are self-described, as opposed to domain specific standards like Bluetooth profiles [14] or UPnP standard device categories [6] for instance. As stated by David Svensson Fors et al. [15], standards need to be exhaustive which is not that easy, even for small applications like controlling a printer.

Granularity of what is a service is dependent of the developer's choice. Nevertheless, one must choose granularity smallest as possible in order to increase reusability and maintainability. Monolithic services are not desirable: high level services will not likely be reusable in other contexts. On the opposite, tiny services, i.e., services providing too basic functionalities, are also a very bad choice as they increase communication schema and debugging cost.

Among dozens of services we developed, we can cite four examples to illustrate our granularity choice: the light controller service which is controls the light in a room, the video service that streams data from a camera over the network, the speech activity detection service that estimates whether the sound from a microphone service contains speech, and the 3D tracker service that inputs video streams from video processing services in order to compute the 3D positions of people in a room (see Section VI-A).

2) Connectors: Connectors are communication ports that can be instantiated by any service to exchange data with other services. Services can have several connectors to logically separate data according to their origin. Each connector is independent from the others. It is identified by a name, a human readable description and a set of sockets where it can be reached.

Connectors can send data over TCP or UDP. In this last case, OMiSCID guaranties (re)ordering of messages. In case of message lost, each peer is notified. Connectors can send data (*input* type), receive data (*output*) or both (*input/output*).

3) Variables: Variables describe the service and its state. A service can expose as many variables as needed. Variables are defined by these attributes:

- *Name*. Name of the variable (254 characters max);
- *Description*. A human description of the variable;
- *Type*. Type is given as a text attribute. It can be used to parse variable value;
- *Access type*. It is possible to define *constant* variable. In case of a constant variable, value of the variable cannot be changed after starting the service. Variables can also be *read only*: modification requests coming from another service will then be automatically rejected;
- *Value*. This attribute contains the value of the variable.

Any service can register to another one to receive notifications when the value of one or several variables changes.

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.

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*

Even if OMiSCID is not limited to these, there are 2 kinds of workflows that are usually mixed:

- A peer to peer approach. After receiving a message from a service and processing it, a response message is sent back to it. Input/output connectors are used in this case;
- A data flow approach. After receiving a message on an input connector and processing it, a message with the result is broadcasted on another output 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 using XML, YAML [16] or JSON [17] format and allows for more advanced operation and extensions (see Section IV-C).

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, particularly in ubiquitous computing environments. It is not uncommon to filter services based on their current state, description or provided functionalities. 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 criteria appears or disappears.

OMiSCID provides the basic logical combination of predefined search criteria (variable value/name, connector properties, etc.). They are implemented as functor (function object). For instance, to search a *Camera* service with an output connector named *data flow* or a service *Encoding* not running on the same computer, one can write the following (C++) filter:

```
Or(And(NameIs("Camera"),
       HasConnector("data flow",AnInput)),
   And(NameIs("Encoding"),
       Not(HostIs(GetLocalHostName()))))
```

It is possible to extend filter capabilities providing more complex search primitives by implementing custom functor objects. Figure 7 and Section VII-C give clues about OMiSCID service discovery capabilities.

D. OMiSCID Gui

OMiSCID provides a simple solution to declare, to discover and to interconnect services. However, in an ecosystem spread with a multitude of services, it becomes a requirement, for both the users and the developers, to have an interface helping the visualization, the monitoring, the interactions and the control of all the services. Additionally, the debugging of a service (or a federation of services) in this *wild ecosystem* can, without appropriate tools, be a painful task. Given this facts, we developed a graphical front-end to OMiSCID: the OMiSCID Gui (see Figures 1 and 2).

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. 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. The service browser also provides an extensible set of contextual operations to be applied on the selected services. Default operations include for example monitoring a connector (watching or sending messages) and monitoring a variable (watching changes or sending modification requests). Among all the extensions available and easily installed using the Netbeans Plugin interface, one can find:

- A simple variable plotter that can dynamically create and display evolution of (numeric) service variables (see Figure 2);
- A family of plugins that allow the display of 2D information such as video stream or custom shapes representing for instance regions of interest of a 2D/3D tracker;
- A plugin that displays a graph of the services present 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 plugins repository already packed with visualization, controls, debugging plugins and can be extended by developers. All Netbeans platform plugin can also be integrated into our platform and vice versa. Its ease of use makes it a must-have tool for OMiSCID development, demonstration and service oriented application development.

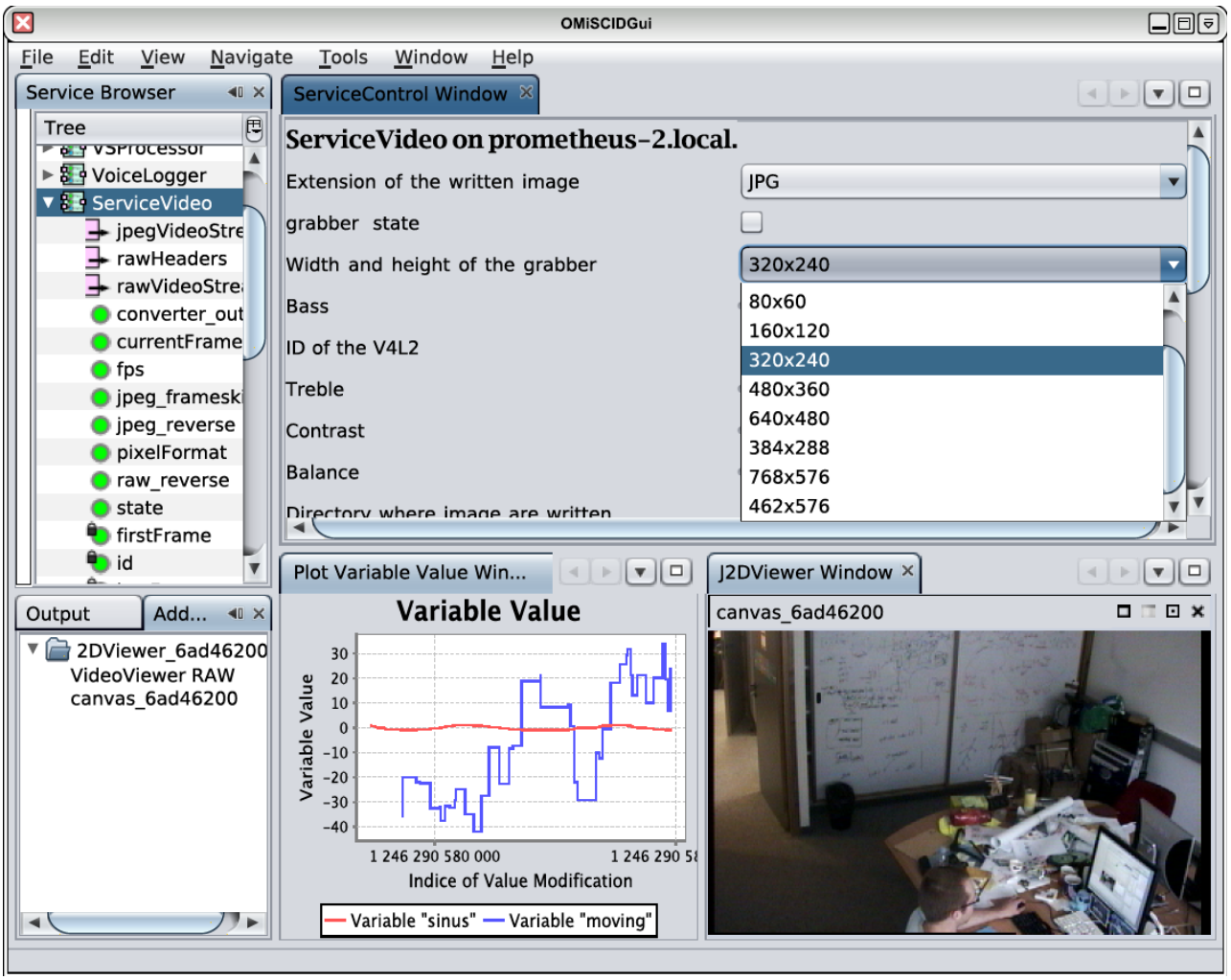

Figure 2: OMiSCID Gui. Public available plugins like running services properties (top left corner), variable value plotter (middle bottom), camera view (right bottom), service video controller (top right) are depicted.

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 (PyMiSCID, note that the Python version used to be a simple wrapper around the C++ one). Moreover, the Java version can be used from Matlab and any other language running on the Java virtual machine (JavaFX script, scala, groovy, JavaScript, etc.) We also provide an OSGi abstraction layer that exposes OMiSCID with standard OSGi paradigm. OMiSCID was developed over a set of guidelines rather than over strict specifications. All the implementations are fully written in the target language, thus ensuring speed, reliability, close integration with data structures 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 provides common system objects like sockets, threads, mutexes, etc. The Java version has been successfully used on portable devices like a PDA and on the Android platform. All implementations can interoperate with each other on any supported platforms.

B. User Friendly API

In order to simplify interpersonal communications between OMiSCID users, we developed a common *User Friendly API*. It was defined to be easy to learn, easy to use and portable in several languages. Indeed, concepts, methods and parameters follow the same API in C++, Java and Python. However, each implementation takes advantage of the language specificities and design patterns. The API

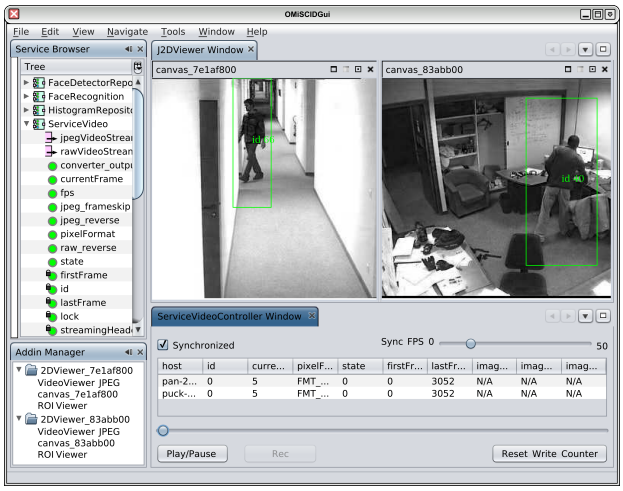

Figure 1: OMiSCID Gui session while monitoring and debugging multi-camera tracking system. Basic OMiSCID Gui functionalities are visible: monitoring of running services properties (top left corner), camera and tracking views (top right, available public plugins), video service controller in order to control 2 video services (right bottom) and the add-in manager to show active plugins (left bottom).

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.

The final *User Friendly API* is designed to be non-invasive. The OMiSCID code, inside a service, must be short, understandable and disjoined of the core source code. For instance, the following is a short example written in C++ that shows how to create a service with an output connector, how to register it over the network and finally how to send data to everyone connected to it:

```
#include <Omiscid.h>
using namespace Omiscid;

// Create a service named camera
Service * pServ = ServiceFactory.Create( "Camera" );
// Add a output connector to it
pServ->AddConnector( "data flow",
                     "images stream", AnOutput);
// Register and start the service
pServ->Start();
// ...
// Send a video buffer to all client via
// the "video flow" connector
pServ->SendToAllClients( "video flow", Buffer );
```

C. New functionalities in OMiSCID 2.0

The current version of the OMiSCID middleware is 2.0. This version brings new requested functionalities: object serialization and remote procedure call.

Indeed, 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. However it has been requested by developers to provide a simpler way to encode and decode textual messages. Thus, OMiSCID 2.0 provides a simple way to marshal and unmarshal any object to a JSON [17] representation. This allows easy communication between services without paying attention to the parsing and serialization of messages. The second improvement is the ability for a service to 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. Again these extensions are cross-platforms, cross-languages and benefit of specific improvement or features depending on the implementing language.

D. Performances

With the intention to provide our community with insight about OMiSCID performances, we conducted several experiments. Each one of these experiments were done on a cluster of computers called Grid5000 [18]. The choice of using a cluster is motivated by the fact that computers are on an isolated network, without any kind of perturbation (e.g., network maintenance, backup in progress, etc.). Moreover, it insure that each operating systems is newly installed, thus without side-effects of having old libraries or an unstable/tweaked system. The last criteria that motivate this choice is that experiments performed on Grid5000 are easily reproducible.

To evaluate the OMiSCID discovery process, we set up dedicated tests performing registering and searching tasks. Regarding the registering operation, at first sight, it is not a costless operation. Indeed we must generate and validate, for each newly created service, a unique service *id* (see Section III-A1) over the network. Our experiment shows that registering 100 services from 3 different computers takes less than 1 second. Regarding the searching task, also referred as lookup task, even if it is a linear process for OMiSCID (search time is linear in terms of number of running services to query), searching involves network communication and thus may takes time. Our experiments reveal that finding a service among more than 400 others, distributed over 4 computers and using a simple variable value takes less than 20 ms long in average. Obviously, performing more specific searches, using for instance user-defined search filter (processing video stream to select a camera for instance) will eventually take more time.

Another performance measurement we performed was the latency introduced by OMiSCID message splitting mechanism (see Section III-B). Those tests were performed between two computers. We compared results using NTTCP —a Linux program that measures the transfer-rate between two computers, and using OMiSCID. Each test was run 1000 times using messages ranging from 512 bytes to 2 megabytes. The result show that the latency introduced by

OMiSCID for message handling is around 4ms in average comparing to usual TCP/IP connections.

The reader might refer to [19] for other tests on performance and scalability.

V. Cases Study

For the past few years, we have used OMiSCID middleware in different research projects [20], [21], [19], [22]. In [19], 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. This stream is simultaneously accessible by multiple services such as visualization services and 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 [22], [21], OMiSCID is used for the implementation 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 [23] 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 [22] 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, which, by extension, ensures a decrease of development time along the years.

In the following sections, we will introduce a 3D video tracker —an OMiSCID redesigned example, some examples illustrating reusability (e.g., a network of Multi-modal Towers and Human Simon Game). The last illustration, present a Wizard of Oz experiment conducted in our lab which show various usage and benefits of using OMiSCID.

VI. Showcase: refactoring, reusability and monitoring

A. Refactoring, the 3D Video Tracker case

To interact with people in smart environment, it is important to be able to detect people as well as to maintain an estimation of their current state e.g., position, speed, posture, etc. We have developed a video tracking system adapted to dynamic and complex environment. The 3D tracking was initially an improvement over an existing 2D tracker. It was running multiple 2D trackers and, by merging the different outputs, was able to compute a pseudo 3D estimation. This tracker had initially a monolithic architecture for performance reason: image acquisition and tracking process were done within a single process.

A full 3D tracker must estimate directly the targets in 3D using information from several cameras. Evolution from 2D to 3D tracking required a change in the architecture as the complexity became too high to run on a single processor. For instance, the number of camera increasing from one to four and sometimes many more cameras. For robustness, reusability and maintainability reasons it is mandatory to split image acquisition and processing software part.

The chosen architecture is shown in Figure 3. This architecture introduces a new concept: service factory. Services factories are services (following the factory method pattern) that are designed to start but also instantiate, on demand, a (parametric) service. Factories can be seen as daemon services, ready to launch other service(s) as to fulfil applications needs. Such scheme eases the deployment of distributed applications. In this example, a Video Service Processing Factory can create a Video Service Processing (VSP) with special pipeline treatments over images from a camera.

The architecture is thus modular and distributed in order to reduce computation time and to avoid costly images transfer over the network:

- A video service is in charge of grabbing images from each camera;
- A video service processor (VSP) factory is attached to each video service. Its role is to initialize one or more VSP in charge of image processing tasks;
- The main tracker program is in charge of targets tracking; it automatically detects all VSP Factories, requests for VSP creation and connects to them.

This refactoring, performed for the 3D tracker, allows us to create lots of software components that are reused later in many other perceptive applications.

Demonstrations of the 3D tracker system are available on the PRIMA channel (see the Tracker, Human Simon Game and more videos on the PRIMA channel [24]).

B. Reusability, the Multimodal-Tower Network case

In the context of the CASPER project (Communication, Activity Analysis and Ambient Assistance for Senior PERsons), a project for maintaining elderly people at home, we developed a multi-modal localization system. This system (see Figure 4) associates on each tower an omnidirectional camera with an array of microphones. Tracking is done combining visual tracker of bodies and acoustic tracker of people when they speak.

Building such application is facilitated by the reusability of already available services: microphone service for audio recording, speech activity detection service, video service for images acquisition, VSP factory (see previous section) for video processing. Only one service was built in this case, the localization service. This service can dynamically register, using filters (see Section III-C), to all data coming from all

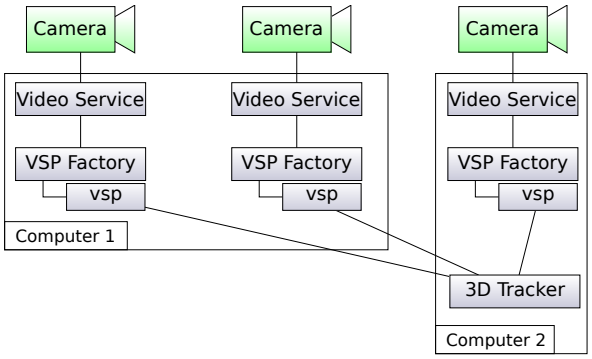

Figure 3: Video tracker architecture. For each camera, a Video Service is in charge of grabbing images and sends them to a Video Service Processor (VSP). This Video Service Processor is instantiated on demand by a VSP Factory. The 3D tracker service automatically finds all VSP Factories, asks them to create VSP and connects to them as to handle targets management.

towers in the room. Then, after an auto-calibration phase, it can track speaking targets in the room.

As it is usually the case, reusability increases maintainability. Using services in many different contexts is a good way to deeply test them in adverse conditions, with new client services. For instance, in the realisation of this system, we highlighted different issues in certain existing services that were not evident to spot before their integration in this particular setting. It is not worth to say that, correcting problems in a service is a benefit for all applications using it.

C. Reusability, the Human Simon Game case

Validating performances of a 3D video tracking system is usually done using well known databases containing annotated recordings providing ground truth labels. The results of the evaluated system are then compared to the ground truth using different kinds of metrics and thresholds in order to provide a score. Such approach are very convenient to validate a tracking system regarding the state-of-the-art (for publication purpose for example). However, generally, the tests are performed off-line and the thresholdings often lead to imprecise measurements. Additionally, test-databases are usually designed to evaluate very specific characteristics and are conditioned as well by very specific environmental conditions. As a result, the outcomes obtained by testing your system over such databases are not necessary representative of the real performance of your system. They might not highlight the particular benefits your system is bringing compared to other existing solution. Robustness to environmental artefacts, such as change in light exposure or random ambient noise for instance, is often omitted. In an attempt to provide an online alternative for the validation of our 3D video tracking

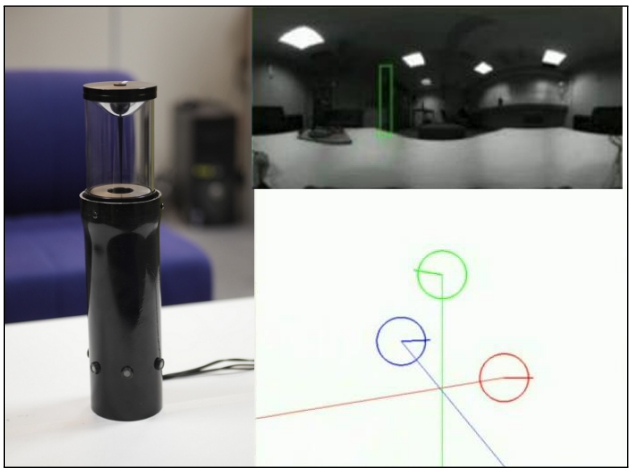

Figure 4: Multimodal tower equipped with an omnidirectional camera and a set of microphones. Multiple towers may be disseminated in the environment and constitute a sensory-network used to follow and infer activity of elderly people at home. One can see the tower itself, a view from one panoramic camera (top right) and the auto-calibration algorithm configuring the relative position of 3 towers.

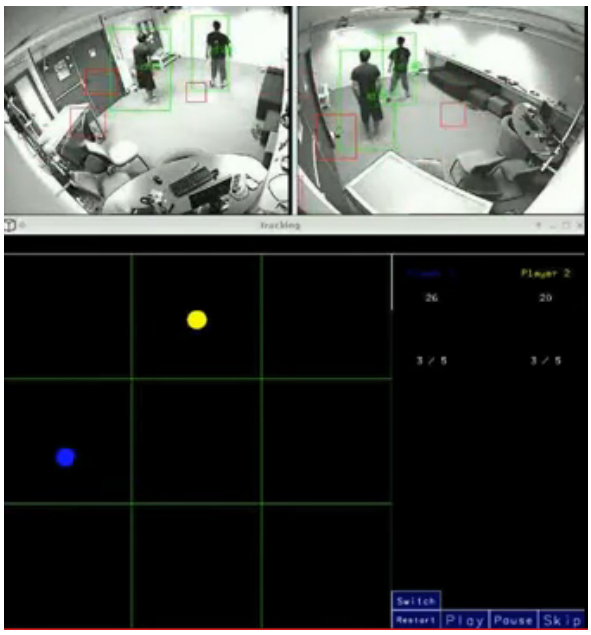

Figure 5: Human Simon Game. Players follow projected instructions on the wall. They play against each other. They must run and squat in cells to activate them. As in usual simon game, the activation sequence is longer each turn[10].

system, we designed an experimental-game showing, in real time, the performance (speed performance, number of false detections, resistance to occlusions, etc.) of our solution in adverse conditions (rapid moves, occlusions, bad lightning condition, etc.).

In order to achieve this goal, we designed a Human Simon Game (see video on the PRIMA channel [24]) based on the famous game for child: one need to remember an audio sequence associated to blinking color buttons. In our case, as seen on Figure 5, we decided to virtually cut the room space in 9 cells, each representing a "blinking button" of the Simon game. Cells can be activated by one or two players squatting and standing back up in it. The game is similar to the usual Simon Game but is adapted for two players: each player has a color and must validate an always increasing sequence of cells by moving and squatting over the virtual 3x3 grid.

In our case, setting up this validation system was an easy task. The 3D video tracker, the posture estimator (for squats), the steerable projector (projecting the game interface on a wall in front of the players) were all existing services that we reused. The only software component developed was the core algorithm of the actual game.

VII. Full case study: Wizard of Oz experiment

In the field of human-computer interaction, a Wizard of Oz (WOz) experiment is a research experiment, in which subjects are confronted with a computer system that subjects believe to be autonomous, but that is actually being operated or partially operated by some hidden experimenter(s): the wizard(s). The goal of such experiments is to study the usability, acceptability and efficiency of a proposed system, functionality or interface —often hypothetic or unfinished— by evaluating the interaction of the subjects with it rather than focusing on the quality of the proposed solution. The advantages of performing a WOz experiment versus actually evaluating the real system or interface are that it saves time and money. Indeed the WOz experiment is a quick and cheap way of (in)validating a set of proposed functionalities without investing resources in a system that might be, first, reconsidered regarding the feedbacks collected from the user experience, and second, just impossible to design due to the absence of required technologies to build it, or the lack of budget and/or time. In the WOz, certain functionalities can be implemented while reconsidering the other. For instance, if we consider the case of smart environments — environment equipped with sensors and actuators designed to provide assistance to user in daily tasks and activities, it can be very annoying and sometimes not an option, to dispose of a perfectly working environment if the only functionality wanted to be evaluated is, for instance, how users manage to undo an action performed in the background by this environment —switching the TV off, turning the music back on, opening the shutter or raising up/down the volume of some other devices. A more time-efficient and money-efficient option would be to focus the resources on the functionalities of interest and manage to fake the other functionality by, for instance, remote controlling the environment by one or more experimenter. In a WOz experiment, the missing functionalities are emulated by an experimenter hidden from the subjects performing the evaluation. 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.

A. Requirements

Even if performing a Wizard of Oz (WOz) remains in many ways more advantageous, depending the context of the experiment, setting up such experiment requires an important preparation. Importantly, additional constraints appear if the settings have to be mobile, i.e., to be carried in different places. The wizard must have access to a multitude of information in order to control the system as well 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 wizards (there might be more than one), also need the proper controllers to remotely manipulate the system. Such a setting requires an extensive use of wireless or wired communication between software components: controllers and controlees. 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. The more reusable the perceptual/actuator software components are, the cheapest and fastest the experiments will be.

B. Experimental Settings

On an ongoing research project [25], 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, are asked to teach the agent to control the environment in order to organize a small meeting. A classic example would be for the subjects to teach the agent to switch on a light when people are entering 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

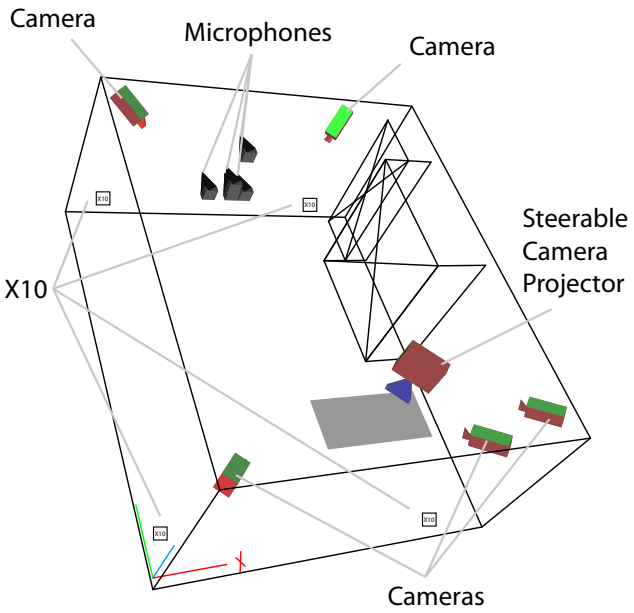

Figure 6: PRIMA's Smartroom. The smartroom is equipped with perceptive equipments (cameras and microphones), and with actuators (mobile steerable camera/projector pair, X10 power controllers). It has been designed to permit immersive user studies with activity recognition (perception and situation modelling) and system feedbacks.

software and a simple user interface. This interface allows collecting real-time feedbacks from the subjects (good, bad) during the session. Alternatively the agent can be embodied by other devices such as, for instance, a robot. In this case, more perceptual components are available for the experimenters and the interaction between the subjects and the agent is different (e.g., the rewards are more natural to provide). The agent, as any other service, connects through the wireless network to a situation modeller service that provides a situation model [21] of the current situation. When requested by the subjects, the agent performs an action in the environment by sending orders to actuator services present in the environment. Subjects can, whenever they agree or disagree, give a positive or negative reward to the agent. With such a settings, the agent learns to control the environment, senses and acts using dynamically discovered services, and finally, learns from the feedback provided by the subjects the correct association between situations and actions.

The *environment* is an office (Figure 6) spread with many actuators and sensors. OMiSCID allows each of them to be accessible and controllable by services all-over the network (Figure 7). Among the sensors we find: cameras, microphones, a thermometer and a 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 actuators, we list: a steerable video projector, some x10 controllers, loud speakers and even windows 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, it was told to the subjects that each chair was mounted with pressure detector to detect when someone sits. Because we had no such device installed in our environmental facility, 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).

C. OMiSCID At Glance

For this experiment we deployed more that 20 services spread on 5 computers running different operating systems (Linux, Windows and MacOSX). Figure 7 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. We next review some of the advantages of using OMiSCID in this experiment:

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 have used different languages. C++ was used for performance reasons 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 or the PanTilt controllers. Java has been used to develop some of the OMiSCID Gui module but also to access the different online web services exposed in the environment such as the weather service. JavaFX was used to develop the wizard control's interface. Its script language makes 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 the *archiver*. Using a service repository, they were able to filter services that were present

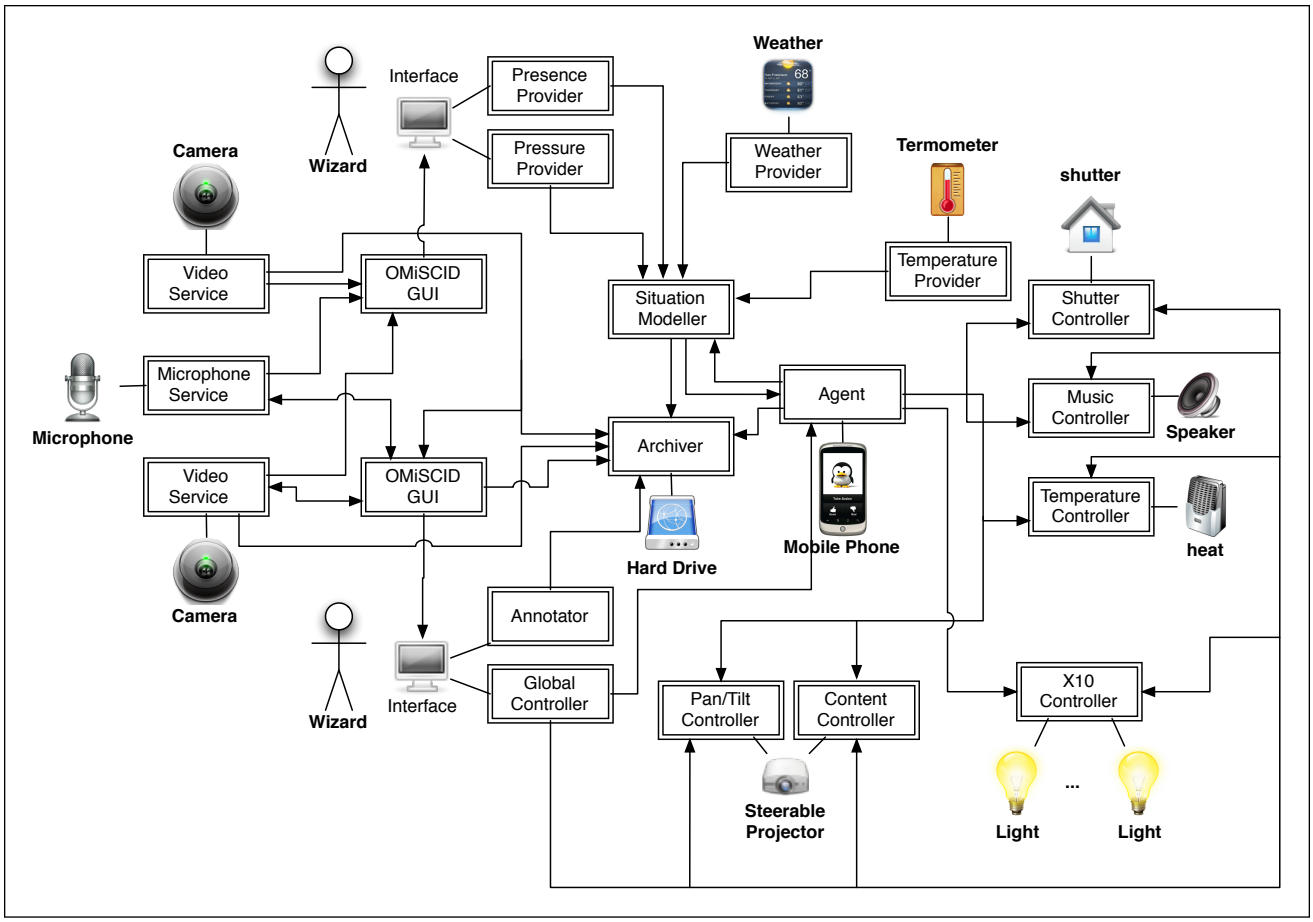

Figure 7: Graph of services for our Wizard of Oz experiment. Services used were mostly created for other experimentations (video services, microphone service, music controller, etc.) and were directly reused and interconnected with specific orchestrating services for this experiment (annotator, global controller).

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 continuously 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 have been used to get 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.

VIII. Conclusion

OMiSCID is an efficient and lightweight solution for the rapid prototyping of Service Oriented Architectures and applications in the context of ubiquitous computing and ambient intelligence. The solution provides a user-friendly

API to declare, to describe, to discover and to interconnect services as well as to manage their communications.

To be attractive, OMiSCID offers to researchers several facilities for ubiquitous computing with its multi-platform, cross-language capabilities and interoperability. The underlying concepts as well as the API are user friendly and directed toward usability, extensibility, reusability and maintainability. In contrast to existing solutions, the API is non-invasive which keeps the developed solution portable to other paradigm if mandatory. OMiSCID can also be mixed or extend other middleware solutions as we did with the OSGi platform. Using OMiSCID for service discovery, one can built an ubiquitous application interconnecting dozens of services. All networking aspects are handle by the middleware as this solution does not rely on the number of computers, operating systems or network configuration. One can concentrate about core services that will orchestrate the full application. As transfer-rate performances are not altered by OMiSCID usage, it is possible to transfer data from a simple integer to huge video streams. Target applications are thus not limited by OMiSCID.

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 of resources and preparations, particularly when realized in smart environments. We have presented how OMiSCID and OMiSCID Gui can greatly reduce development time, maximizing reusability of existing software, and eases redeployment.

IX. Acknowledgement

For their past work on OMiSCID and/or experimentations depicted in this article, the authors would like to thank (in anti-chronological order) Wafa Benkaouar, Matthieu Langet, Jean-Pascal Mercier, Julien Letessier and Sébastien Pesnel.

References

[1] R. Barraquand, D. Vaufreydaz, R. Emonet, and J. Mercier, "UBICOMM 2010 paper Case Study of the OMiSCID Middleware: Wizard of Oz Experiment in Smart Environments," in *The Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, Florence, Italy, 2010.

[2] 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.

[3] "OSGi Alliance," accessed 17-January-2012. [Online]. Available: http://www.osgi.org/

[4] 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.

[5] 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.

[6] "Referenced specifications UPnP forum," accessed 17-January-2012. [Online]. Available: http://upnp.org/sdcps-and-certification/standards/referenced-specifications/

[7] "OW2 chameleon," accessed 17-January-2012. [Online]. Available: http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose

[8] 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.

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

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

[11] 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.

[12] 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.

[13] "OMiSCID Forge," accessed 17-January-2012. [Online]. Available: http://omiscid.gforge.inria.fr/

[14] "The official bluetooth SIG member website | specification: Adopted documents," accessed 17-January-2012. [Online]. Available: https://www.bluetooth.org/Technical/Specifications/adopted.htm

[15] D. Svensson Fors, B. Magnusson, S. Gesteg\aard Robertz, G. Hedin, and E. Nilsson-Nyman, "Ad-hoc composition of pervasive services in the PalCom architecture," in *Proceedings of the 2009 international conference on Pervasive services*, ser. ICPS '09. London, United Kingdom: ACM, 2009, p. 83–92, ACM ID: 1568213.

[16] "YAML on Wikipedia," accessed 17-January-2012. [Online]. Available: http://en.wikipedia.org/wiki/YAML

[17] "JSON Website," accessed 17-January-2012. [Online]. Available: http://www.json.org/

[18] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jégou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet, "Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform," in *6th IEEE/ACM International Workshop on Grid Computing - GRID 2005*, Seattle, USA, 11 2005.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] "PRIMA Channel on Youtube," accessed 17-January-2012. [Online]. Available: http://www.youtube.com/user/PrimaChannel

[25] R. Barraquand, P. Reignier, and N. Mandran, "The Sorceress of Oz," in *Workshop for Pervasive Intelligibility, part of the Pervasice Conference*, San Francisco, USA, 2011.