Towards Ubiquity in Ambient Intelligence: User-guided Component Mobility in the HI^3 Architecture

A. Paz-Lopez[1], G. Varela[1], J. A. Becerra[1], S. Vazquez-Rodriguez[1], R. J. Duro[1]

[1]*Integrated Group for Engineering Research*
University of A Coruña
Ferrol, Spain

Abstract

Ambient Intelligence (AmI) systems need to be as transparent as possible, that is, their users should perceive only the effects of the features presented to them and, in some cases, some kind of interface. They should not be conscious of how these features are implemented, from a hardware or from a software point of view.

In order to obtain such a high degree of transparency, it is necessary for the system to be able to provide its services regardless of location, adapting to the environment and the context in general and to the available hardware more specifically. This is known as ubiquity, and to achieve it requires considering many aspects, from security and privacy to system interoperability. This paper is mainly focused on the development of two elements related to ubiquity: physical migration of components between different platforms together with their associated runtime state, and the adaptation of those components to the destination platform and physical environment.

These solutions are being addressed within our efforts for the development of a general-purpose middleware for Ambient Intelligence in the framework of the HI^3 project.

Keywords:
ambient intelligence, ubiquitous computing, mobility, service collaboration, smartphone

1. Introduction

One of the main goals of every Ambient Intelligence (AmI) system is to provide a seamless experience to their users. This goal is achieved in AmI systems by offering their functionality to users in the most transparent way possible, and transparent operation has multiple implications. For example, the interaction with the user should use natural interfaces as much as possible, the system should show autonomous and proactive behavior, it should require little intervention in system management and its functionality must be ubiquitous to the users.

Ubiquity, namely the capability of the system to provide its services regardless of location and

Email address: gervasio.varela@udc.es (G. Varela)
URL: http://www.gii.udc.es (G. Varela)

the devices available, is an important topic in AmI, as it is a key enabler to release the user from the management of its applications and data. But ubiquity is not an easy goal and a lot of issues must be addressed. Interoperability between the different systems involved, fault-tolerant distributed operation, physical mobility of system components, self-organization of components, heterogeneity of devices and services available in each environment, user location tracking, and a lot of security and privacy issues, are among the most important ones.

In the framework of HI^3 [1], an integral Ambient Intelligence platform that has been in development in our lab for the last few years, this paper is focused mainly on the mobility of components and some of its associated issues, like reorganization of components and renegotiation and adaptation of behaviors after a component migration process.

Component mobility has been addressed using tools and techniques from many different fields. The distributed computing world has provided general solutions for service-oriented systems and multi-agent systems, but special emphasis must be placed on more specific solutions coming from the area of ubiquitous computing.

Multi-agent technologies are a prominent source of solutions for software mobility because of the benefits obtained by agents with the ability to migrate from one host to another [2]. The most popular multi-agent platforms have some level of integrated support for agent mobility [3] [4]. Commonly, the solutions applied by these technologies are simple, leaving in the hands of the developers a lot of responsibility, but they are also very general, so they can be used for multiple purposes. They provide only code and state migration support, and the main problem addressed is the interoperability between heterogeneous platforms, especially when dealing with platforms that have communications compatibility but are implemented with different base technologies (programming language, computer platforms, etc.). This problem has been recently addressed in [5] by specifying a common interface to manage the lifecycle of agents in a FIPA [6] platform, as well as by providing common agent models and data encoding mechanisms to enable interoperability between platforms that do not support the same programming languages.

Even though they are not directly related to ubiquitous computing, Fluid Computing [7] [8] techniques are interesting proposals for user application mobility. They rely mostly on application state replication and synchronization, especially bearing in mind disconnection issues. In the case of [8] they also provide support for application code replication.

Another important source for mobility solutions is the service-oriented field. The SOA approach has been widely used by ubiquitous computing solutions, and as such, many have recognized the necessity to provide support for service mobility. Two approaches that need to be mentioned are [9] and TaskShadow [10]. The former has extended the OSGi platform to support service replication and state synchronization between replicas. It has also increased the service description capabilities of OSGi with environment contextual information, so that the migration process can be driven by context changes. TaskShadow applies a very different approach to mobility because it does not rely on code migration. Instead, it introduces the concept of task as an abstract description of user intents and a set of functionalities required to perform some user task. It achieves user mobility migrating a user task by searching local services that fulfill the requirements of the task.

The Aura [11] project is in some ways a precedent of TaskShadow. It also contemplates the idea of user task and relies on local components to fulfill the requirements of a task. Aura was designed for a classical desktop environment and thus it does not use context information to adapt

the user task to a new environment.

Finally, another very clear example of mobility solutions is the Gaia OS project [12]. It provides a development framework that divides applications using the MVC pattern. The different components of the application can be migrated and replicated to different hosts.

These projects are mainly focused on the distribution and migration of applications that make use of distributed devices (sensors, actuators and others) available in each environment. Therefore, even though they address important areas of application mobility, like the physical migration of the application, the discovery of compatible devices and some sort of component reorganization, they do not take into account specific topics of AmI applications, like user preference management, adaptation of behavior to the users and the environment, or context management.

We have been working on the development of a complete component mobility solution for AmI applications and its integration in the HI^3 architecture. The final objective is to propose an integral solution to the mobility issues that affect AmI systems, providing approaches for topics like physical component migration, negotiation and discovery of hardware requirements, component self-organization, component self-adaptation to user and environment preferences, user data privacy and component behavior security, as well as application interaction adaptation. This is a long-term development effort and this paper deals with some of the first developments in this area of HI^3.

Specifically, the mobility solution presented here adds to the HI^3 framework the capability of migrating selected services and applications, including their current state, from one platform to another that is located anywhere. Migration is driven by a user mobile device, such as a smartphone, where the user transparently carries information about his/her applications and services. This information includes descriptions, requirements, preferences and needs. As the user goes from one environment to another, the mobile device searches for platforms capable of running the user's components and negotiates the migration of these components with the target platform. Additionally, in order to facilitate the adaptation of migrated components to the new platform, this solution includes a semantic service description model and a matchmaking algorithm. They allow the definition of components in terms of their requirements in such a way that these requirements can be satisfied later at run-time by selecting the best services of the target platform to serve to the migrated components.

The rest of the paper has been structured as follows. Section 2 is devoted to a detailed exploration of the mobility issues and benefits in an Ambient Intelligence environment. A brief description of the HI^3 platform is provided in section 3, along with an explanation of the solutions developed, referencing the previously described use case example. Next, section 4 presents an example use case of the mobility capabilities of the HI^3 platform. A comparison of solutions for software mobility, focusing on those projects related to smart environments or ubiquitous computing, and including the HI^3 approach, is commented in section 5. Finally, section 6 presents some conclusions and future work.

2. Challenges and Objectives

As briefly stated in section 1, ubiquitous access to AmI systems is an important requirement to achieve the transparent operation goals of Ambient Intelligence. Ubiquity implies a lot of

benefits for the users, but it requires some complex requirements to be fulfilled [13]. This section presents the main benefits of ubiquity, and its associated requirements and challenges.

Sometimes, the benefits of migrating components instead of remotely accessing services and devices to achieve ubiquity may not be obvious, because in many cases similar functionality can be achieved while maintaining the different elements of the system distributed in their original hosts. But migration support can improve AmI systems in aspects such as autonomy or network dependency.

In a realistic scenario, the user cannot expect that any platform provide services with the functionality he needs. Furthermore, even if the services of the target platform provide the requested functionality, it is possible that they cannot remotely access to the user data. In these cases, it seems a better solution to migrate some user components from one platform to another, instead of only trying to make use of the available local services as other approximations do.

Moreover, if all the components are executed locally within the data sources and end devices, the network bandwidth requirements are effectively reduced and the network latency is eliminated. This last point is especially important for interactive real-time systems like AmI applications, which must control devices to interact with a human populated environment. Thanks to migration, AmI systems can even continue operating in environments that do not have a network connection, or when the network fails.

Regarding privacy and security, by migrating components, instead of letting some external software access and manage user data, the user components, with their user defined security and access control measures, directly manage the devices and data. As opposed to this, if the service was operating remotely, either local sensing data would be sent to a remote platform, or personal information about the user would be sent to the local services, both of which could imply a privacy problem. However, the migration of software components and data to different platforms may also involve severe security and privacy problems. One of the main concerns is ensuring that malicious components are not able to affect the normal operation of the target platform.

With this in mind, the main goal of this work is to build the basis of a mobility framework that takes full advantage of migrating software components between different platforms without limiting the use of other approaches when appropriate. Thus, the proposed solution can be considered a hybrid approach, consisting of using existing local services when it is possible and migrating user components if existing local services cannot fulfill the user's requirements.

Furthermore, this solution follows a user-guided approach. That is achieved through the use of a user's component running on a user's personal device that has information about the user's preferences and drives the migration process. This approach to performing migrations, as opposed to the traditional method where the platforms monitor the users and actively make all the decisions about migration, increases user privacy. Unknown (to the user) remote platforms do not access user information in order to decide whether a migration is needed, rather, it is software running in the user's device the one that takes the initiative and accesses the information from the different platforms and makes the decision about migration. This way, it is possible to incorporate information about platform trust to the decision process.

However, as we introduced before, this approach comes with a lot of challenges that AmI systems developers must address:

- **Code transfer and state synchronization.** The code of the components of a system needs to be transferred from one host to another, and the receiving host must be able to execute it

[14] [8]. Also the component state must be transferred and recomposed at the destination.

- **Resource allocation.** The destination host must have the resources (computing, sensing, interaction, etc.) required by the component to operate [15] [9]. Furthermore, the migrated components can run into conflicts with previously existing components on the use of a resource.
- **Security and privacy.** The system must safeguard the host environment from potentially malicious incoming components, for example, restricting the actions they can perform. And vice versa, protect incoming components from malicious hosts, so that components perform the actions they are supposed to [13].
- **User interaction.** The user interface components benefit most from mobility, as they need to use local devices and show low latency operation. The main challenges faced in this case are the heterogeneity of the interaction devices available in each environment, as well as the differences in interaction abilities of each user [16] [17]. Ubiquitous UIs must support multiple modes of interaction, as well as autonomous adaptation to the devices and interaction modes available for each environment and user.
- **Component relations managing.** Components are not used in an isolated manner [10]. In general the users are carrying out many tasks at the same time, and the system can even be autonomously carrying tasks out for the users. The system must be able to manage the relations between the components and the tasks in order to optimize the migration process.

As can be seen, Ambient intelligence platforms and middleware have in mobility an important source of challenges and opportunities to address in order to provide developers with powerful tools that facilitate the development of real ubiquitous applications.

3. Component Migration and Adaptation in HI^3

This section is devoted to the description of the particular mobility solutions developed and their integration within the HI^3 architecture for AmI systems.

First of all, a brief description of the HI^3 architecture is presented. This architecture provides support for the development and integration of applications and services in AmI environments. Additionally, this architecture provides the necessary concepts and tools to support other requirements such as mobility, connectivity and ubiquity.

Figure 1 shows the conceptual structure of the architecture. It is a layer-based design that enables the division of system elements into levels, reducing the coupling between modules, thus facilitating abstraction and the distribution of responsibilities. The uniform device access layer (UniDA [18]) is in charge of providing homogeneous and distributed access to the physical devices. The sensing and actuation services layer provides services that are virtual representations of sensors and actuators in the physical environment, hiding part of the complexity of the real devices. The service layer is populated by components that provide shared functionalities to other services or applications. These two service layers include tools to facilitate the development and execution of services, including a service repository, semantic service definition capabilities, service composition facilities or service discovery mechanisms. The applications layer is the

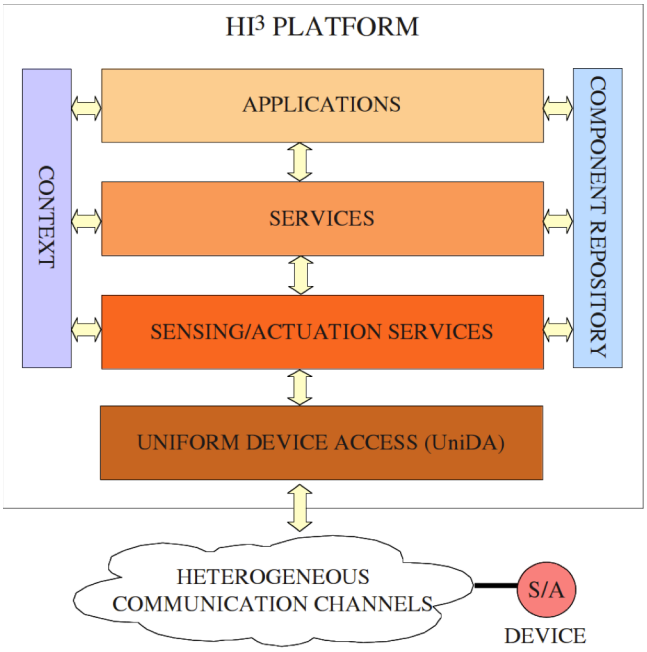

Figure 1: HI[3] software architecture

highest-level layer and the one that hosts the elements representing and implementing applications that provide particular functionalities a user expects from the system. Finally, the context is a common access component for the three higher-level layers of the architecture. Its main objective is to represent the current state of the environment.

In order to implement the previously presented conceptual model, a multi-agent technology based approach supported by the JADE agent framework was chosen. The components of the higher level layers in the HI[3] architecture are implemented as collections of agents that collaborate for distributed task resolution. This implementation of HI[3] has been structured as displayed in Figure 2. It provides a container for AmI services and applications with features such as high level inter-agent communication facilities, multi-agent models with support for the declarative definition of components, development utilities and distributed management tools. This middleware promotes the division of large and complex AmI systems into highly-decoupled components that dynamically and autonomously collaborate to solve complex tasks.

A more detailed description of this architecture and multi-agent platform can be found in [1] [19].

3.1. HI[3] Mobility Subsystem

As indicated above, the proposed mobility solutions are implemented on top of the HI[3] architecture and multi-agent platform. In this platform every component is a collection of agents considered as an atomic element that can migrate. This is a different approach from what is usually done in generic multi-agent platforms, as they often take agents as their atomic components. This decision was made because in HI[3] every component (application, service or sensor/actuator) is a highly-coupled collection of agents that collaborate to offer a very particular

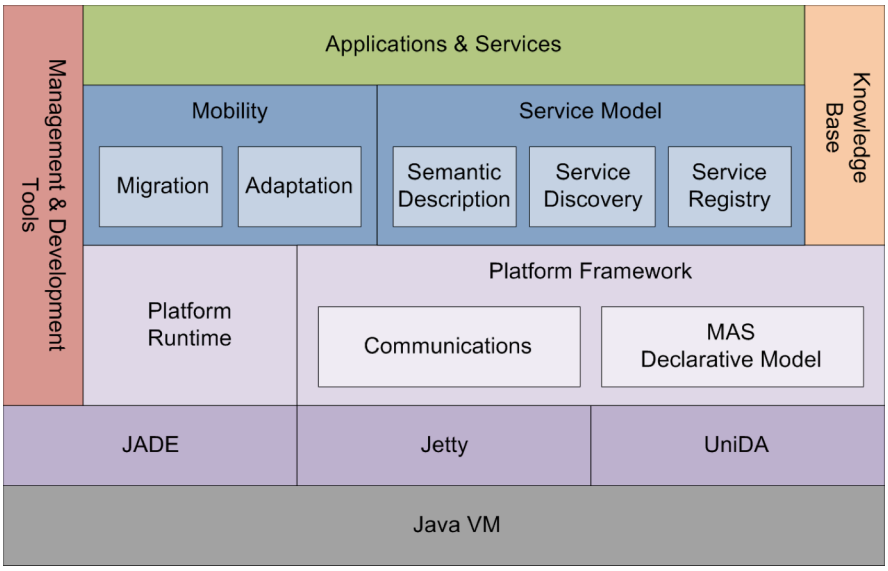

Figure 2: HI[3] multi-agent platform.

functionality. Consequently, it will generally be a good idea to keep them together in the same physical platform.

HI[3] follows a hybrid approach in terms of mobility. Even if all components have integrated support for migration, we are not expecting every component to be migrated; only those that make sense and will obtain some kind of benefit from a migration. This last point is within the responsibility of developers, but components that require interaction with users or the environment are usually good candidates, as they can benefit from being executed in a place that is physically near the hardware or users they have to interact with.

Figure 3 displays a simplified class diagram of the structure of the HI3-mobility subsystem. It shows the main elements of the subsystem and their relationships in the component migration process. This procedure is divided into two main sub-processes. On one hand there is the physical migration of the component from one instance of the HI[3] platform to a new one, and on the other, the adaptation of the migrated component to the new platform and physical environment. This procedure is complemented by the use of a mobile device to drive the migration process. It stores information about the user and his associated components, and proactively searches for valid platforms to deploy the components.

The physical component migration process is mainly managed by the *Mobile User Agent* (MUA), the *Component Mobility Manager* (CMoM) and the *Component Migration Manager* (CMiM). The MUA hosts description information about user associated components, and searches for adequate platforms to deploy them. The MUA uses the CMoM to access information from the guest platforms, such as available services or resources. When the MUA detects a valid platform, it contacts the host CMoM to request a migration of some components to a guest platform. The CMiMs of the two platforms are in charge of managing the transfer of component code and state.

The adaptation of the migrated components to the new environment is mainly managed by the components themselves, but the system provides them with some helper capabilities. Service

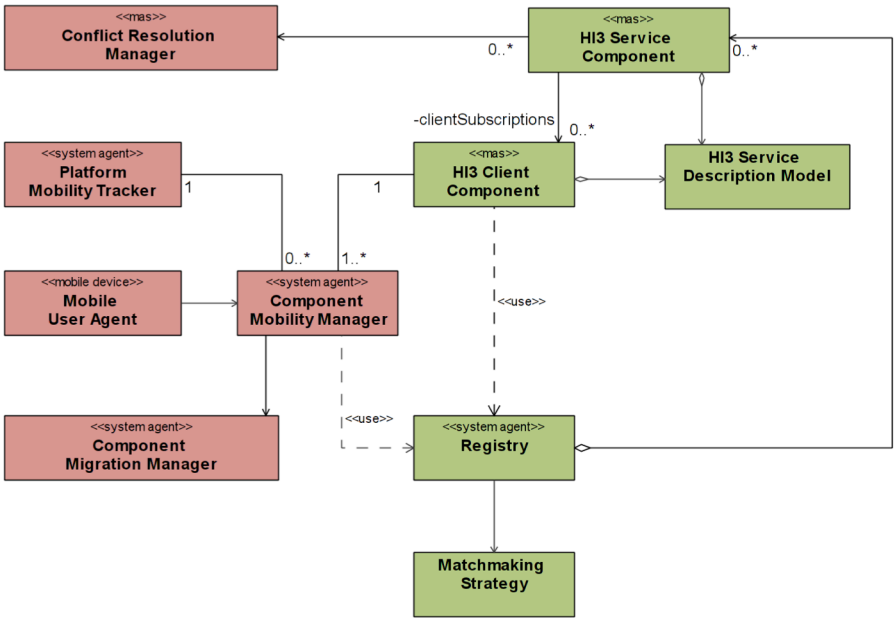

Figure 3: HI[3] mobility subsystem simplified class diagram

components are catalogued in the system registry according to their functionality. Thus when a component arrives in a new platform, it can use the registry to find components in the new local platform that provide the functionalities they require. Furthermore, components can parameterize the functionality to request some specific behavior, and the registry will provide the service that best matches the desired behavior.

The following subsections describe in detail the main aspects of the HI3-mobility subsystem. First, a solution based on a user mobile device to drive the migration of components is presented. Next, the details of the component migration strategy are shown. Finally, a solution for component adaptation to the target environment, grounded in an autonomous service composition framework, is described.

3.2. User-guided Component Migration

The Mobile User Agent is deployed on a mobile device owned by the user, currently an Android smartphone, and stores a reduced view of the user profile and the description of the components associated to him. It uses this information in order to search for appropriate platforms for the user components. When found, it commands the CMoM to migrate the component to the new platform. In the current implementation the user specifies this command, but it is a future work to extend the MUA with autonomous capabilities to select destination platforms.

The MUA is a central element in the HI[3] mobility process. It drives the component migration process by selecting what components to transfer and where they must be transferred. By delegating this responsibility to a user owned device, the users benefit from an increased level of privacy.

In a classical AmI setup, the guest platforms available in each environment would be the ones in charge of monitoring the user state and deciding what to move, when and where. Namely,

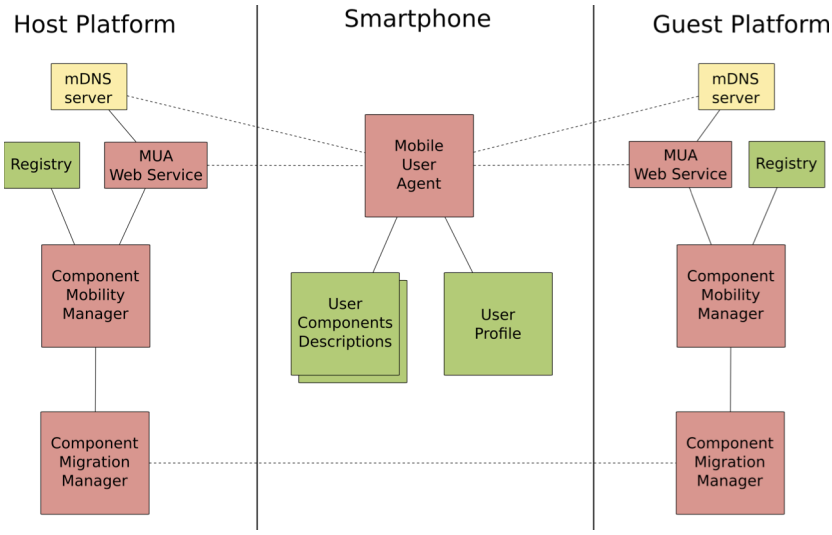

Figure 4: Mobile User Agent block diagram

those remote and unknown platforms would have to access user information in order to be able to make this kind of decisions and this may imply a privacy leak for the user. When using the MUA the tables are turned, and it is a user controlled software element the one that accesses information about the guest platforms, which are only able to access user information if the MUA decides so.

Currently the MUA is implemented as an Android service and application. Figure 4 shows a block diagram of the MUA and its interactions with the host and guest platforms.

Once the smartphone is connected to a WiFi network, it uses the mDNS protocol to discover the available HI^3 platforms. As the MUA is not implemented using multi-agent technologies, due to implementation constraints in Android, it uses a RESTful web service to interact with the CMoM of the guest platform. The web service has two responsibilities. On one hand, in guest platforms it allows access to platform information such as available resources and functionalities, or software version. This information is matched with the user components and profile information, and it informs the user about what platforms are most suitable for each component. On the other hand, in the user host platform, it allows the MUA to access reduced views of the user information provided by the CMoM, so that it can be stored by the MUA.

Regarding its implementation, on the platform side, the web service is implemented using J2EE and the Metro web service stack. It is deployed in a Jetty HTTP server embedded in the HI^3 platform which allows every HI^3 component to present web interfaces implemented using J2EE. On the client side it uses the Spring Android library for accessing RESTful web services. Finally, as indicated before, in order to discover the available platforms, it uses the mDNS protocol by searching for a new service type, *hi3-mobility*. In the client side, it uses the *jmDNS* JAVA library, and the platform is announced using the *avahi daemon* in a GNU/Linux host.

3.3. Component Migration Process

When a CMoM receives a request to migrate one or more components to a remote platform, it relies on the CMiM (there is one for each HI^3 platform) for the migration. The CMiM must

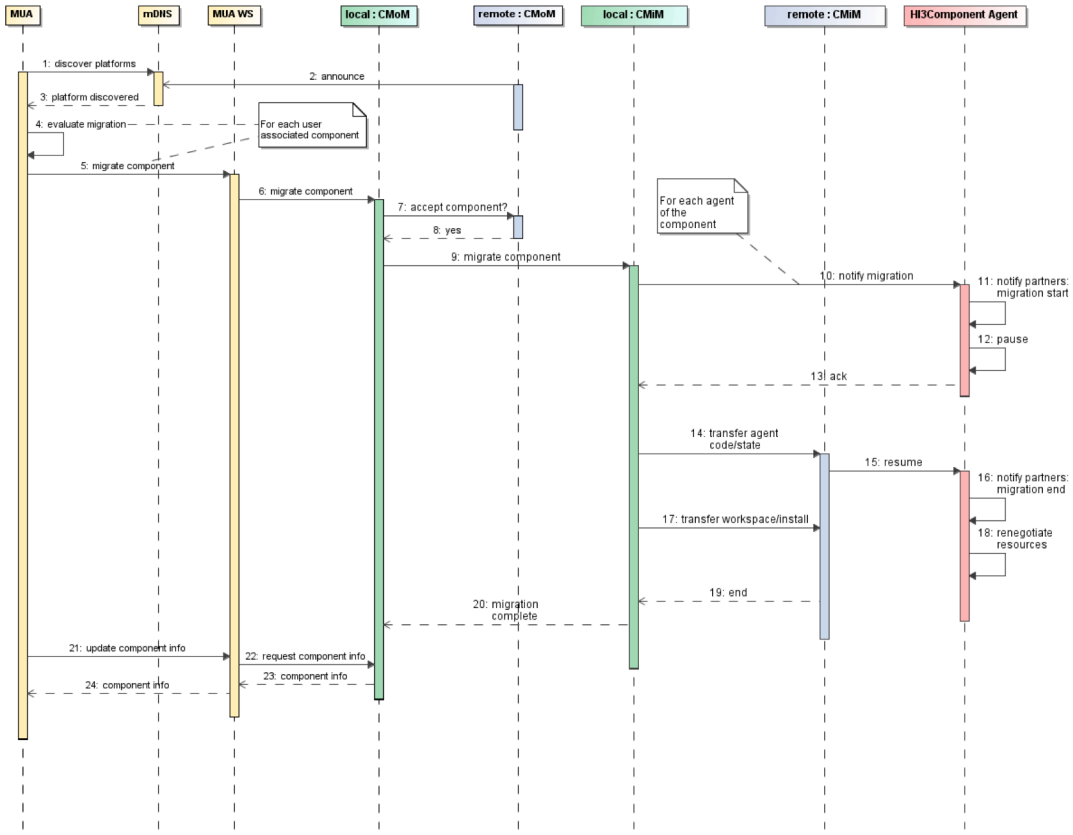

Figure 5: HI3 component migration process

transfer the three elements that make up an HI^3 component at runtime. The *workspace*, which is a place where the component can store persistent data, the *install*, which is basically the component code, required libraries and configuration files, and the *runtime state* of the component.

The JADE agent platform, on which the HI^3 multi-agent platform is based, has integrated support for agent mobility. A JADE platform is divided into agent containers, so that one platform can have multiple containers in physically different hosts, and it supports agent mobility between containers. Unfortunately all containers depend on services provided by a main container, leading to the presence of a central point of possible failure. Furthermore, every container must be managed by the same administrators, so it is not possible to migrate components between platforms managed by different entities, which is a basic requirement in the case of ubiquitous computing and AmI.

There exists an extension of the JADE platform to support inter-platform agent mobility [5], but it couldn't be used in HI^3 because it relies on the system class loader to obtain the code of the agents. HI^3 uses private class loaders for each component, instead of having a single global class loader as in the case of default JADE, as a way to enforce code access privileges. This is important, not only due to security reasons, but also because it allows components to use different versions of the same code (for example libraries) without conflicting. This is especially important in open systems like those designed for AmI and even more when mobility is supported and

components from different authors and users should coexist.

The migration process is basically implemented by the CMiM. As represented in Figure 5, the CMiM starts by requesting the remote platform to accept the new component. If the remote platform accepts the component, it answers with metadata about the platform, so that the local platform can check whether it matches the basic requirements of the component (mainly computing resources). Then it notifies the start of the migration process to every agent of the specified component. Each agent notifies its partners that it is going to be temporarily paused due to a migration and it then prepares its workspace for the migration and pauses its execution.

When all component agents are paused, the CMiM uses JAVA serialization to serialize the state of each agent, and packs this state with the code of the classes directly used by the agent (extracted by introspection of the agent fields, methods, declared superclasses and interfaces). The state and code of each agent is sent to the new platform which then resumes the execution of each agent.

Once the agents are running in the new platform, the local platform compresses and sends the component workspace and installation directories containing the complete code of the component, as well the libraries and other files it requires. This transfer is carried out asynchronously and, thanks to the classes included with the agent state, agents can start running before having the complete component code.

While the few classes sent with the agent state may be enough to execute some simple agents, in the case of complex ones, they would need access to more code even before the component installation transfer is complete. To solve this problem, the class loader of the transferred agents is able to dynamically request classes from a remote CMiM (the CMiM of their previous local platform). The CMiM will check if the requesting agent is one of the migrated agents of the component holder of the requested code, and will pack and send the class code.

Once the component install transfer is finished, the remote CMiM informs the local CMiM of the successful deployment of the new component and the end of the process. Finally, the local CMiM stops the execution of the transferred component and removes its agents (which were previously paused) from the platform.

3.4. Service Description Model

Taking a look at the overall migration process, the convenience of automating and generalizing the process of restarting the execution of the migrated components on the target platform can be immediately appreciated. Thus, the system developed here provides a common model and a set of facilities the migrated components need in order to adapt their configuration and tasks to the new execution conditions. Specifically, the mobility system manages the process in charge of discovering the most appropriate local services that provide the functionalities the migrated component needs for its execution. This whole process is supported by the semantic description capabilities provided by the HI^3 Service Description Model (HI3-SDM), which facilitates the development of self-describing modular components, which can be published, automatically located and invoked across the platform.

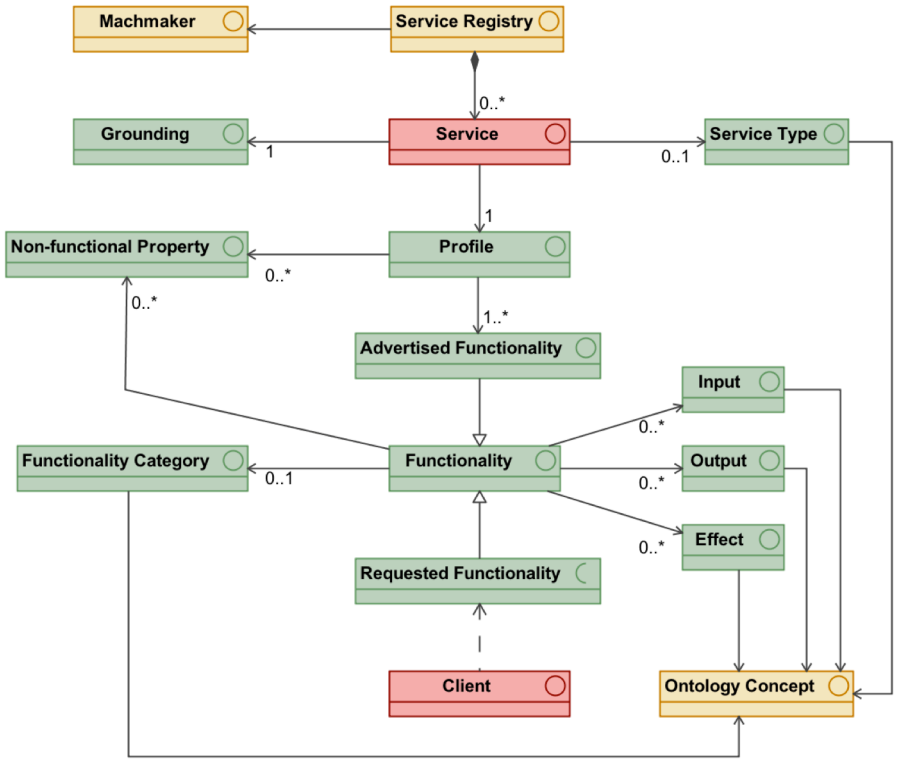

Figure 6: HI[3] Service Description Model (HI3-SDM)

The ontology based service description approach is, nowadays, the most promising solution for supporting autonomous service discovery and service composition processes [20] [21]. This approach specifies services and their properties using a common model (ontology), enabling the participating entities to reason and match service concepts. Currently, there are several ontology-based languages for describing services, such as the Web Service Ontology (OWL-S) [22] and the Web Service Modeling Ontology (WSMO) [23]. Furthermore, there exist a large number of well-established service discovery protocols and service communication protocols that makes the adoption of a single technology quite an unrealistic strategy, particularly in the broad and heterogeneous field of Ambient Intelligence. Thus, interoperability between different service providers and clients, without forcing them to use a particular technology, can be accomplished using a common service description model that enables mapping among the heterogeneous service description languages. This is the solution adopted in the HI3-SDM, which proposes an abstract model inspired by the AmIi service description model [24] [25].

As shown in Figure 6, the HI3-SDM supports the specification of both functional and non-functional (i.e., QoS and security) service properties. The central element of the HI3-SDM is the *functionality* concept, which represents the description of any capability that can be advertised by a service or requested by a client. This description is expressed in terms of the information transformation produced by the service (*inputs* and *outputs*), the state change produced in the environment (*effects*) and the *functionality category*. These functional properties are defined referencing existing ontology concepts.

In order to realize the abstract model presented in Figure 6, a complete service model

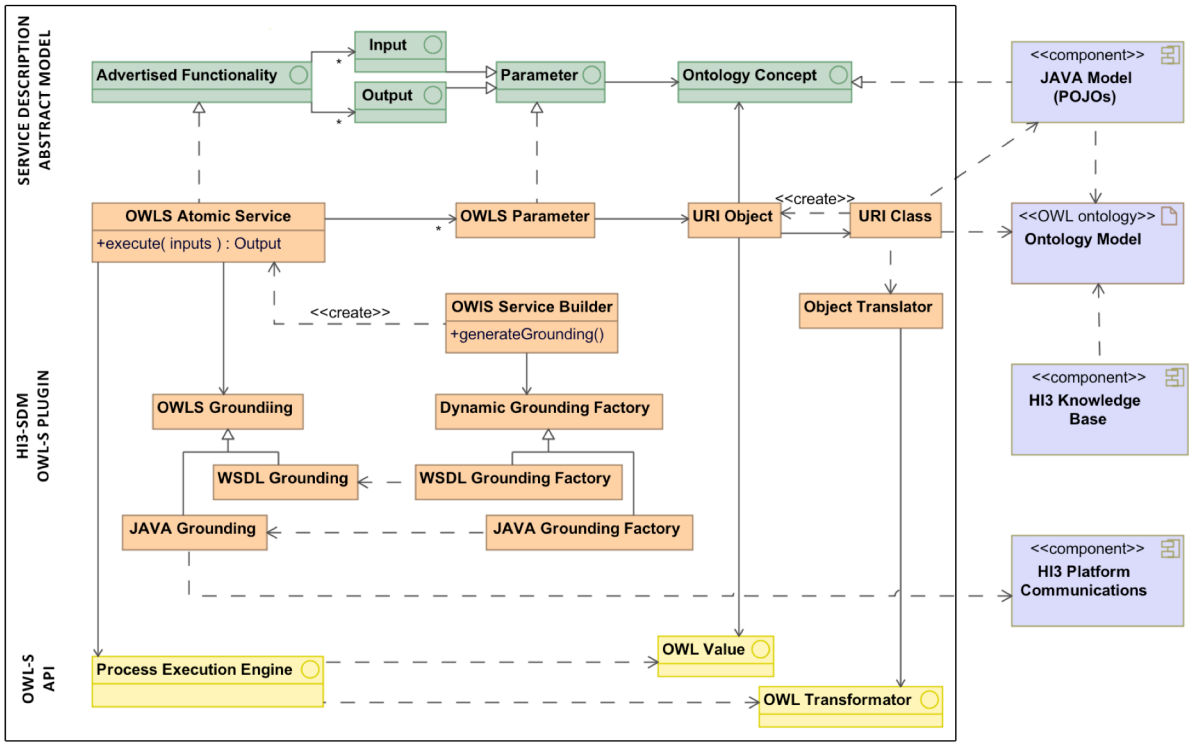

Figure 7: HI[3] Service Description Model implementation grounded on OWL-S

implementation based on the OWL-S ("OWL for Services") language was developed. OWL-S provides similar concepts to those defined by the HI3-SDM, and divides the description of services into three main aspects, "what a service does" (service profile), "how the service is used" (process model) and "how to interact with the service" (grounding) [26]. Figure 7 displays some design aspects of the implemented solution. The proposed approach abstracts and automates the tedious process of specifying the grounding of the semantic services. Specifically, this implementation provides a strategy to dynamically generate a Java Grounding that enables the automatic mapping between Java objects and OWL ontology concepts. Furthermore, this Java Grounding allows for the transparent invocation of a remote service using the communications capabilities of the underlying HI[3] agent platform.

Finally, another important component of the HI3-SDM is the *matchmaker*. The objective of the matching process is to compare requested functionalities to semantically described service advertisements stored in the service registry using inference rules enabled by ontologies [27]. As a result of the matching process, a service that matches the requested functionalities, expressed in terms of inputs, outputs and effects, can be selected and transparently executed by the runtime environment.

3.5. Negotiation and Conflict Resolution Strategy

Once a component arrives at a new destination, it uses the HI[3] Service Description Model and the system registry to find partner components to collaborate in task resolution. The arrival of new client components with different requirements could lead to conflicts with the current

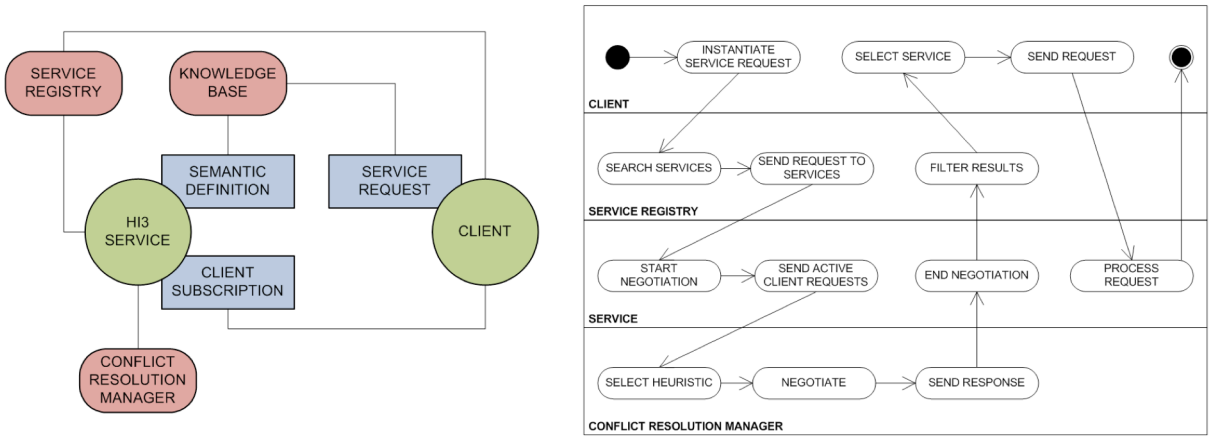

Figure 8: HI^3 service negotiation and conflict resolution process

operation of local components. Therefore a component request negotiation and conflict resolution system, was introduced.

Figure 8 summarizes the main elements of this system that are in charge of managing the adaptation of the migrated components to the new platform and environment. In this process, when a migrated component arrives at a new platform, it can use the registry of the platform to find services that provide the same *functionality* it was using before the migration process. For this purpose, the migrated component creates parameterized *component requests* that semantically describe the desired functionalities using the HI3-SDM. Additionally, each component has a collection of clients that can store a list of previous active parameterized requests (subscribers) and incorporates capabilities for managing conflicts in the access to its functionality. Thus, when a new component requests an action and there can be a conflict with some resources managed by the service, they have integrated capabilities that solve possible conflicts with other currently active clients subscribed to the service. The detailed sequence of actions for this negotiation process is shown in the activity diagram of Figure 8.

The parameter variables of each request designate different semantics about the desired action. These parameters are specified by the HI3-SDM and their meaning is dependent on the service implementation. By default, every request has an *importance* parameter, which designates how strong the commitment of the client is to the action requested. This parameter is used by the service in order to solve possible conflicts between different client requests. Services can instantiate a conflict resolution manager that includes different strategies to solve conflicts. In the current implementation only one strategy is available that support the resolution of conflicts using the *importance* parameter and a fuzzy inference system implemented with the jFuzzyLogic library.

These strategies constitute a first step towards the autonomous adaptation of systems to the changing environments envisioned in fields like ubiquitous computing, ambient assisted living, ambient intelligence or human-centered computing. The proposed negotiation and conflict resolution strategy provides HI^3 components with a series of mechanisms that allow them to adapt to new environments, with the restriction that every environment must share a common semantic description model. Nevertheless, to achieve a complete solution it is necessary to improve these

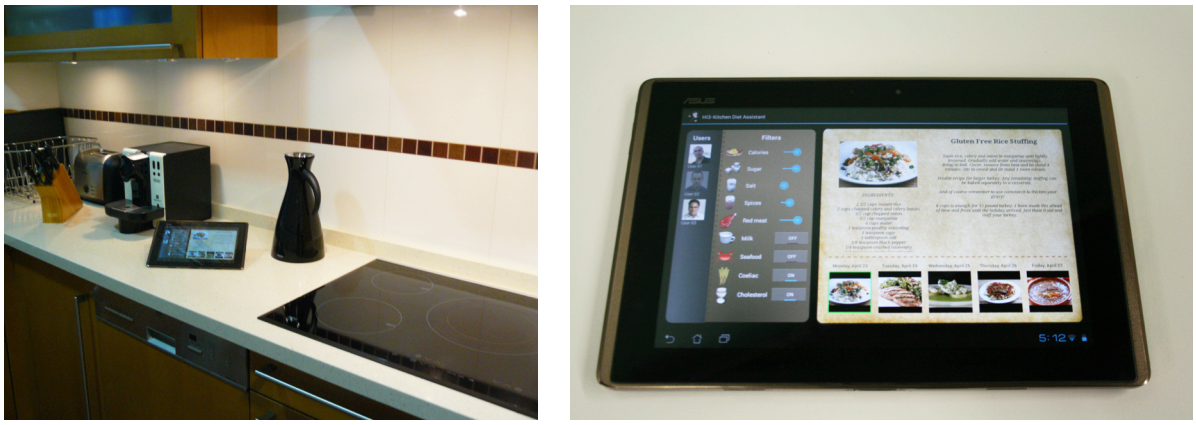

Figure 9: Photographs of the kitchen and tablet used for testing the developments of the project

strategies with characteristics such as interoperability among heterogeneous systems, computational load balancing, user and environment data privacy safeguard measures or capabilities to preserve the functional integrity of the whole system.

4. Use Case Scenario for Component Mobility

In order to better illustrate the inner workings of the presented solution, the current section will show an example use case of the mobility capabilities of the HI^3 platform.

The example has been implemented within the framework of an ongoing project we have for the innovation in the kitchen space. The project goal is to increase the level of automation of the kitchen and its associated tasks, by taking advantage of high-end appliances and new mobile user devices, like tablets and smartphones. One of the developments of the project encourages families to follow a healthier lifestyle. It includes a Meal Planner application for Android tablets that relies on a HI^3 service, which uses a catalog of healthy recipes and information of the family members (personal and health information), to create a balanced and varied plan of meals for each day of the week.

For a mobility use case scenario, we have imagined a case in which the family is receiving a visitor for some days. Therefore the Meal Planner should bear in mind the new member and adapt the meal plan to his/her requirements. More specifically, we have thought out a case where the grandfather or grandmother, who lives in a retirement home, is coming to his/her daughter´s home for the holidays.

The retirement home has implemented an Ambient Intelligence system that aids the caregivers in their daily chores. Among other things, this system allows the medical staff to manage the diet of the residents and to monitor the residents using different devices. The system uses an instance of a Resident Agent (RA) service component for each resident. This service is in charge of managing the health information of the resident, providing it to other services (for example the catering service in order to prepare the meals), as well as interacting with the different monitoring devices associated to the resident.

In order to keep the example as simple and relevant as possible, we have imagined a scenario where the resident is diabetic and uses an insulin pump with a glucose sensor. The RA, when

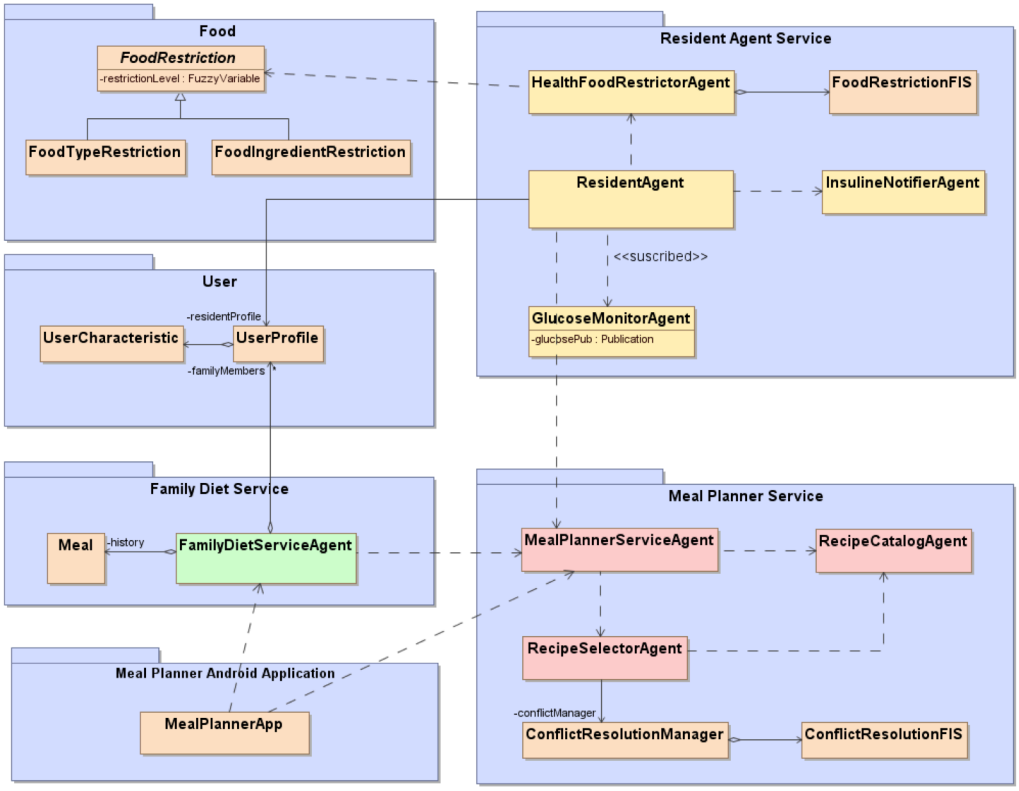

Figure 10: Use case example service components

running in the resident home system, on one hand interacts with the glucose sensor in order to obtain information about glucose levels and report it to the medical staff. On the other hand, it interacts with a Meal Planner service used by the catering staff to plan the meals for the residents. The RA service is designed as a mobile HI[3] component, so, when the grandmother/grandfather arrives at the family home, he/she can use a mobile device (currently a smartphone) to transfer his/her personal services from the resident home to the family home. Thus the RA can continue interacting with the glucose sensor, and can use local available services, like the Meal Planner, in order to adapt the system and environment to the user.

Figure 10 shows a simplified view of the internal architecture of the components involved in this use case scenario. As can be seen, the two main elements of the example are the Meal Planner service and the Resident Agent service.

The former is available in the family home AmI system and provides meal-planning capabilities to the tablet application. It is composed of three different agents: a coordinator agent that interacts with external entities and coordinates the other two, a recipe catalog agent that manages a database of recipes and a recipe selector agent, that uses information about the family members in order to plan an appropriate and balanced selection of meals. The user information is received in the form of food restrictions, for example, if some member of the family should avoid salted meals or has some kind of food allergy. As the information coming from different users can lead to conflicts, it uses a conflict resolution manager to select the most important restrictions

among the conflictive ones. This conflict manager relies on a fuzzy inference system, implemented with the jFuzzyLogic library, to solve the conflicts.

Food restriction information can arrive to the MP from multiple sources and users. One example is the Family Diet (FD) service, which interacts with the Android application, allowing the family members to specify their restrictions. Another source of food restriction information would be the RA service when the grandfather/grandmother is in the family home. As indicated in section 3.5, these requests are parameterized by including a fuzzy variable indicating the importance of each request for a specific user. Therefore, depending on the health state of each user, his/her associated request would specify a different level of importance, which would be used by the previously cited conflict resolution manager of the MP to solve possible conflicts.

The RA is composed of four different agents. A coordinator agent, a glucose monitoring agent that relies on a glucose report agent to create reports about the glucose levels of the resident, and a food restriction agent that uses a fuzzy inference system to automatically generate a set of food restrictions using health (diseases, allergies, etc.) and personal information of the resident. The RA is designed and implemented as a moveable HI^3 component, thus it contains specific logic to control the operation of the component before and after a movement takes place. Not all HI^3 components are designed to support mobility, as the developers of the system decide which ones must require mobility, while the others would work remotely.

When the grandfather/grandmother arrives at his/her family home, he/she uses the smartphone to start the transfer of personal services to the local available AmI system. The smartphone is automatically connected to the family WiFi network, which is previously known, and, as specified in Figure 5, it uses mDNS to find the available HI^3 containers. When it finds a trusted one, it interacts with the resident's home AmI system through the MUA Web Service to request the transfer of the user's moveable services, in this case the RA, to the family home system. At this point, the resident home system notifies the different agents of the RA that they are going to be moved to a new location. The agents prepare themselves for the transfer by closing/freeing local resources. In the proposed example, the food restrictor agent will save its currently generated restrictions for use at the destination, and unload the fuzzy inference system, in order to alleviate the transfer payload. Once they have finished, the agents are paused and the transfer process begins.

As shown in Figure 5, the transfer is carried out in two different phases. First, the agents' states and directly used code is packaged compressed and transferred. After this, the agents are already available at the destination system, and their behavior execution is resumed. This behavior could be slightly limited, because not all the component code and files are available yet. Once the agents are restarted at the destination, the source platform starts the transfer of the complete component installation directory, which includes all the component code, library dependencies and data files.

When the agents are restarted at the destination, the system executes a callback in each agent to notify them that they have been transferred to a new destination. Agents can execute the required logic to adapt to the new environment. In the example, the glucose monitor agent will look for a glucose sensor device and interact with it. The food restrictor agent would search for a service that offers meal planning functionally, and send a request to replan with the resident specified restrictions and parameter. Then it would wait until the complete component directory is transferred in order to again load the fuzzy inference system, which relies on files that would not

Local network (Gigabit Ethernet)		Internet (300 Kbit upload)	
First stage (~200 bytes)	Second stage (~7.5 Mb)	First stage (~200 bytes)	Second stage (~7.5 Mb)
2500-2700 ms	14000-15500 ms	4000-5000 ms	~300000 ms

Table 1: Transference times measured for the Resident Agent component

be transferred alongside the agent's code and state.

This kind of decisions, about what is transferred, and how the agent behavior is modified when running in a remote platform relies on the developers, and can be of paramount importance in order to obtain short response times when transferring components. Table 1 shows the transfer times we have measured for this example. Once the first stage is finished, the agents are already in execution at the destination. The decision to unload the fuzzy inference system, thus delaying the requirement of the jFuzzyLogic library, allows us to considerably reduce the component state in order to reduce the time that the component agents are paused.

To finish this discussion about a possible use case scenario where ubiquity is achieved by mixing component mobility with remote access, we want to stress some of the benefits that could be obtained by transferring some user components, like the RA, to a platform physically close to the user:

The RA requires interaction with local available hardware, in this case the glucose sensor. By transferring the monitoring software to a local platform it can interact directly with the devices, for example using a ZigBee or Bluetooth network, without requiring any other intermediary, and effectively reducing the delays in accessing the device, which can be of paramount importance for other devices/purposes, like user interaction hardware.

Even when a remote system could rely on locally available services to access devices, it could be the case that the local system does not have installed a component with the required *functionality*. By having the ability to move software components from one platform to another, we maximize the possibility that users can access their utilities wherever they are.

Even though this was not the case in this particular example, the RA could have used locally available information, like temperature, domotic devices state, etc. This information is private to the family, and sending it to a remote platform for processing could imply a privacy leak.

Privacy leaks could also happen from the other side. For example, sensitive information about the user that is transferred within agent states to remote platforms. This case could be minimized or even completely solved by programming the agents not to transfer sensitive information. For example, the RA could decide not to transfer the user health information with the food restrictor agent, and transfer only the previously generated food restrictions, which contain far less sensitive information about the user.

Even if in the current implementation an internet connection is required for component transference, once transferred, they can continue operating without a network connection to the original platform. This could be useful for mobile environments like vehicles. And as future work, it is planned to allow the possibility of transferring the component state and code to the smartphone that would store them until a valid destination platform is found to restart the execution of the component.

5. Mobility Solutions Comparison

This section relies on the information extracted from the related work presented in section 1 in order to provide a comparison of the different approaches for component mobility. Obviously, to compare the different solutions to each other and to HI3-mobility, a comparison framework must be established. Here it was based on a set of properties that were taken as important characteristics that every mobile software solution should bear in mind. The characteristics are presented below and they are followed by a comparison of the different technologies along with a discussion about these results.

The main characteristics or features considered in the comparison were:

- **Code migration.** The capability of the solution to physically migrate components. It avoids the need of having the components installed on every platform and allows the user's own applications to be used wherever he is.
- **State migration.** Related with the previous characteristic, it refers to the ability of the solution to move the state of the applications so that they can continue operating even without network connection.
- **Security and privacy.** Mobile security and privacy measures implemented by the solution.
- **Environment Adaptation.** This category aggregates all those characteristics related to the adaptation of mobile applications to the new environments while migrating.
  - *Runtime environment*. Capability of the solution for determining if the destination platform complies with the execution requirements of a component (platform software version, memory, CPU, etc.)
  - *Service needs*. The ability to connect the migrated services to the most adequate local ones. That is, find those local services that better fulfill the requirements of the migrated services and adapt their behavior to the limitations of the available local functionalities.
  - *Contextual adaptation*. This characteristic refers to an extension of the previous ones, where the platform, user and environment situation information is used in order to optimize the adaptation of the migrated components to a local platform. For example, information about shared resource usage, user preferences or needs, etc.
  - *User interaction*. This is a specialization of *contextual adaptation*, and represents the ability of the solution to adapt the user interface of the system to the interaction capabilities available in an environment.
- **Component distribution.** Related to most of the characteristics of *environment adaptation*. It refers to the ability of the solution to select the destination of the migrated components using the previously described adaptation characteristics. This includes dividing the components of a service or application if necessary.
- **Task awareness.** Applications and services are not used in an isolated manner, they are grouped in order to provide solutions to particular tasks the user is actively involved in, or

	Agents	Aura	Fluid C.	GAIA	HI3-mobility	Preuveneers	TaskShadow
Code migration	Yes	No	No	Yes	Yes	Yes	No
State migration	Yes	No	Yes	Yes	Yes	Yes	Yes
Security and privacy	Low	Low	Low	Low	Low	Low	Low
Runtime environment	Yes	Yes	No	Yes	Yes	Yes	No
Service needs	No	Low	No	Medium	Medium	Medium	Medium
Contextual adaptation	No	No	No	No	Medium	Low	Medium
User interaction	No	Low	Low	Low	Low	Low	Low
Component distribution	No	Low	Low	Medium	Medium	Low	Medium
Task awareness	No	Medium	No	No	Low	No	High

Table 2: Comparison of component mobility solutions

even tasks that are proactively performed by the system in the background.

These nine characteristics cover many important requirements for software mobility, especially in the Ambient Intelligence or ubiquitous computing field. Therefore they can be a good first evaluation set to establish a reasonable framework to compare mobility solutions for Ambient Intelligence. Table 2 compares the technologies reviewed in section 1 in terms of these characteristics.

The discussion of these results can be started considering agent mobility solutions. They are the simplest ones, focused exclusively on code and state migration, they do not provide support for component adaptation and other features. This leaves a lot of responsibilities on the hands of developers, who would have to deal with all the associated issues mentioned above.

Two of the solutions, Aura [11] and TaskShadow [28] [10], share many similarities. Both use the idea of user task as a central concept, and they also take a similar technical approach. They do not migrate code and rely exclusively on available local applications and services that comply with the task requirements. The main difference between them is their application field. Aura was designed for a traditional software environment, while TaskShadow was designed for smart environments, so it provides support for context-based selection of the services, thus being a better option for AmI systems.

Fluid computing solutions, like IBM's fluid computing [7] or FlowSGI [8], are focused on the synchronization of the application model (from a MVC point of view), so they do not implement migration or adaptation strategies, and rely on local applications (selected by the user) to this end. Thanks to its application state synchronization capabilities, fluid computing is good for multi-user environments, but the lack of component adaptation features limits its usage in AmI environments. Its approach is very different from HI3-mobility, sharing only the ability to migrate the runtime state of the components. They are especially focused on multi-user applications with synchronized states and not really on application mobility.

From a more general point of view, GAIA [12] [29] is a good example of mobility support. It

fully supports code and state migration, it provides capabilities to describe application requirements and search other services that match them, and it supports component distribution. Nevertheless, it does not support the use of contextual information to drive the adaptation to new environments, nor does it support task awareness.

Finally, the service-oriented approach of [9] supports service replication and state synchronization between replicas. It also uses contextual information in order to allow for a more user and environment adapted migration of the services. Therefore, even though it does not support task awareness, and uses only environment sensing information as context, it is one of the most complete approaches for mobility in ubiquitous computing or AmI environments.

The HI^3 approach shares many similarities with GAIA and the Preuveneers approaches, but it elevates contextual information to a more prominent position, providing capabilities to solve shared resources and preference conflicts between users. HI^3 also provides a complete ontological framework for service discovery and request parameterization, and includes basic support for task awareness, by grouping and managing several user services at the same time.

With respect to Aura and TaskShadow, even though the latter makes use of contextual information to drive the migration, the approach to mobility chosen by HI^3 is more flexible. Aura and TaskShadow do not support code migration, which is a core feature in HI3-mobility. This feature allows users to continuously use their own applications and services, while relying on local services and devices to access local resources or improve performance. TaskShadows relies exclusively on locally available resources, thus if there is no resource that provides one of the required functionalities, the tasks cannot be moved.

It is worth noting that, currently, even though the HI3-mobility approach has devoted some efforts to increasing user privacy, most solutions have done very little in terms of the security and privacy aspects of mobility. It is also important to point out that user interaction adaptation is future work for all of them.

Summarizing, most efforts in the line of incorporating mobility to AmI systems are still in their infancy. After being focused on the physical migration of components and state synchronization, many solutions have started to explore the adaptation to new environments by including contextual information in their service discovery strategies. However, there are still a lot of issues that have been barely addressed such as privacy and security, resource and user preference conflicts or user interface adaptation.

6. Conclusions

Ubiquity is key characteristic of Ambient Intelligence systems. It increases the level of transparency by allowing access to system functionality independently of the user location, effectively releasing users from the management of their applications and data. Component mobility is a prominent approach to achieve ubiquity, as it provides important benefits in areas such as system operation latency, system autonomy or user data privacy. However, these benefits are usually overlooked by the majority of AmI projects, which rely exclusively on distributed operation to achieve ubiquity.

Software mobility is a complex topic but, when applied to AmI and ubiquitous computing environments, its complexity increases even more, as a whole lot of new issues related with interoperability, system autonomy, device access or security and privacy are introduced. This

paper describes the support for component mobility that has been integrated in a general-purpose ambient intelligence development platform called HI^3 architecture. Unlike other approaches, the solution presented here pays particular attention to the preservation of user privacy and preferences, and uses this information to drive the component migration process.

A user-guided approach, on the one hand, enables the mobility system to take advantage of information on user preferences to optimize the matchmaking between the user software requirements and the capabilities advertised by a platform. On the other hand, by limiting the exposure of user private data to unknown third parties, it allows the minimization of privacy problems associated to user software mobility.

Concerning the physical migration of components, the HI3-mobility approach adapts existing approximations to the particularities of an AmI multi-agent based platform, paying special attention to code access privileges within the platform, as well as latency and fault-tolerance issues during the mobility process.

Regarding component self-adaptation, which is also an important and studied topic, the solution proposed allows components to select the most appropriate services in the target platform according to the changing execution conditions, and helps components in solving conflicts from its interaction with multiple users. A lot of work is still required in order to obtain a complete mobility solution for AmI. From extending the current developments, for example with autonomous and intelligent selection of target platforms based on non-functional requirements (i.e., QoS and security), to new developments in areas like protection of the mobile components from their hosts and vice-versa, user interface adaptation after a migration process or data privacy and synchronization.

References

[1] A. Paz-Lopez, G. Varela, J. Monroy, S. Vazquez-Rodriguez, R. J. Duro, HI3 Project: General Purpose Ambient Intelligence Architecture, Proceedings of the 3rd Workshop on Artificial Intelligence Techniques for ambient Intelligence, AITAmI08 (2008) 77–81.

[2] D. Lange, M. Oshima, Seven good reasons for mobile agents, Communications of the ACM 42 (3) (1999) 88–89.

[3] Jade. Java agent development framework.
URL http://jade.tilab.com

[4] Aglets. Java mobile agent platform and library.
URL: http://aglets.sourceforge.net

[5] J. Cucurull, R. Martì, G. Navarro-Arribas, S. Robles, J. Borrell, Full mobile agent interoperability in an IEEE-FIPA context, Journal of Systems and Software 82 (12) (2009) 1927–1940.

[6] FIPA. Foundation for intelligent physical agents.
URL http://www.fipa.org

[7] D. Bourges-Waldegg, Y. Duponchel, M. Graf, M. Moser, The Fluid Computing Middleware: Bringing Application Fluidity to the Mobile Internet, The 2005 Symposium on Applications and the Internet (2005) 54–63.

[8] J. Rellermeyer, flowSGI: a framework for dynamic fluid applications, Ph.D. thesis (2006).

[9] D. Preuveneers, Y. Berbers, Pervasive services on the move: Smart service diffusion on the OSGi framework, Lecture Notes in Computer Science 5061 (2008) 46–60.

[10] G. Pan, Y. Xu, Z. Wu, S. Li, L. Yang, M. Lin, Z. Liu, TaskShadow: Toward Seamless Task Migration across Smart Environments, IEEE Intelligent Systems 26 (3) (2011) 50.

[11] J. P. Sousa, G. D, D. Garlan, Aura: an architectural framework for user mobility in ubiquitous computing environments, Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture 25 (August) (2002) 29–43.
[12] M. Roman, H. Ho, R. Campbell, Application mobility in active spaces, in: 1st International Conference on Mobile and Ubiquitous Multimedia, Oulu, Finland, 2002.
[13] X. Song, U. Ramachandran, Mobigo: A middleware for seamless mobility, in: Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on, IEEE, 2007, pp. 249–256.
[14] J. Cucurull, R. Martì, G. Navarro-Arribas, S. Robles, J. Borrell, G. Suades, Fragment Transfer Protocol: An IEEE-FIPA based efficient transfer protocol for mobile agents, Computer Communications 33 (18) (2010) 2203–2214.
[15] U. Aguilera, A. Almeida, P, D, Continuous service execution in mobile prosumer environments, in: IV International Symposium of Ubiquitous Computing and Ambient Intelligence, UCAmI 2010, 2010, pp. 229–238.
[16] L. Balme, A. Demeure, N. Barralon, J. Coutaz, G. Calvary, Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces, in: Ambient, Vol. eds, 2004, pp. pp291–302.
[17] D. Thevenin, J. Coutaz, Plasticity of user interfaces: Framework and research agenda, Proceedings of INTERACT'99.
[18] G. Varela, A. Paz-Lopez, J. Becerra, S. Vazquez-Rodriguez, R. Duro, UniDA: Uniform Device Access Framework for Human Interaction Environments, Sensors 11 (10) (2011) 9361–9392.
[19] A. Paz-Lopez, G. Varela, J. Monroy, S. Vazquez-Rodriguez, R. J. Duro, HI3 Project: Software Architecture System for Elderly Care in a Retirement Home, in: 3rd Symposioum of Ubiquitous Computing and Ambient Intelligence, 2008, pp. 11–20.
[20] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-Oriented Computing: a Research Roadmap, International Journal of Cooperative Information Systems 17 (02) (2008) 223.
[21] J. M. Rodriguez, M. Crasso, A. Zunino, M. Campo, Improving Web Service descriptions for effective service discovery, Science of Computer Programming 75 (11) (2010) 1001–1021.
[22] OWL-S: Semantic markup for web services.
URL http://www.w3.org/Submission/OWL-S/
[23] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, D. Fensel, Web service modeling ontology, Applied Ontology 1 (1) (2005) 77–106.
[24] S. Ben Mokhtar, N. Georgantas, V. Issarny, COCOA: COnversation-based service COmposition in pervAsive computing environments with QoS support, Journal of Systems and Software 80 (12) (2007) 1941–1955.
[25] N. Georgantas, V. Issarny, S. Mokhtar, Y. Bromberg, S. Bianco, G. Thomson, P. Raverdy, A. Urbieta, R. Cardoso, Middleware Architecture for Ambient Intelligence in the Networked Home, Handbook of Ambient Intelligence and Smart Environments (2010) 1139–1169.
[26] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. L. McGuinness, E. Sirin, N. Srinivasan, Bringing Semantics to Web Services with OWL-S, World Wide Web Internet And Web Information Systems 10 (3) (2007) 243–277.
[27] A. Gal, P. Shvaiko, Advances in Ontology Matching, in: T. Dillon, E. Chang, R. Meersman, K. Sycara (Eds.), Advances in Web Semantics I, Vol. 4891 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, Ch. 6, pp. 176–198.
[28] G. Pan, Y. Xu, Z. Wu, L. Yang, M. Lin, S. Li, Task Follow-me: Towards Seamless Task Migration Across Smart Environments, IEEE Intelligent Systems.
[29] C. Hess, R. Cerqueira, A. Ranganathan, Gaia: A Middleware Infrastructure to Enable Active Spaces, IEEE Pervasive Computing 20.