A flexible framework to prototype, develop and manage ubiquitous applications for intelligent environments

Matheus Erthal*, David Barreto*, Douglas Mareli* and Orlando Loques*
*Institute of Computing
Federal Fluminense University
Rio de Janeiro, Brazil
{merthal, dbarreto, dmareli, loques}@ic.uff.br

***Abstract*—Due to recent advances in mobile computing and wireless communication technologies, we can see the emergence of a favorable scenario for building ubiquitous/pervasive applications. This work proposes a state-of-art framework for building those applications by solving some of the main issues of the field, namely the heterogeneity of devices, the amount and diversity of context information and services, and the difficulty of developing and testing ubiquitous applications. The abstractions provided, and the management services conceived, allow developers to handle resources spread within the intelligent environment in a simple and homogeneous way. Furthermore, they enhance the capability of implementing context-aware applications, while still keep the flexibility on developing such applications. In order to demonstrate the applicability of the proposal, we implemented the concepts in a platform named *SmartAndroid*, which contemplates most of the framework's features and has enabled the construction of a set of example applications to validate the framework. Above the platform we implemented a management and prototyping interface that uses a tablet device to allow the testing and evaluation of applications running on different configurations of the environment. This intelligent environment may be populated with a mix of simulated and real devices, what may reduce the costs of prototyping. The interface also enables the end-user programming of context rules through a high level user interface.**

I. Introduction

The idea of Ubiquitous Computing (UbiComp), nowadays widely discussed, was first proposed by Mark Weiser in the 1990's [1]. Also referred by pervasive computing, the field aims at providing a different paradigm of human-machine interaction. As traditional applications provides services to users through their explicit commands (using devices as mouse, keyboard, monitor, for example), in ubiquitous applications (UbiApp) the interaction happens without the need of explicit interaction, i.e. the application tries to discover the users needs through the acquisition of context (using sensors) and the knowledge of their preferences, and providing services in the environment (using actuators). These environments enhanced with sensors and actuators to provide automated services to persons are also called Intelligent Environments (IE).

The construction and manipulation of UbiApps represent major challenges for developers, especially in terms of technical knowledge required and the availability of real devices during application development. Some of these challenges can be well highlighted: (i) there are difficulties in establishing a common protocol for communication between the components of the distributed system, because of the *heterogeneity of devices* involved, (ii) the interactivity of UbiApps is hampered depending on the amount and *variety of context information and services* available in the environment, (iii) *developing and testing applications* require high availability of resources, such as sensors (e.g. presence, lighting, temperature), actuators (e.g. keys, alarms, smart-tvs), including new embedded devices, or physical spaces, such as a house for applications of type smart home.

In this work we propose a framework for developing UbiApps in IE. The goal is to provide support for the programming, testing and execution of applications, thus allowing to deal consistently with systems great complexity. The framework stands out for addressing the challenges already identified in UbiComp [2]. The *heterogeneity of devices* is handled through the definition of a Distributed Component Model, that provides abstractions to encapsulate these devices, also called resources, enabling developers to interact with them seamlessly. Regarding the *variety of context information and services* issue, we propose a solution for context interpretation that allows developers to create and manage context rules at runtime, and users to set their preferences in the IE. Finally, regarding the issue on the *resource availability*, the framework includes the Prototyping Interface for Pervasive Applications (PIPA), focused on visualization and testing UbiApps, mixing real and virtual components.

The concepts of the framework were materialized on a platform called *SmartAndroid*[1], whose development has enabled appraisal in order to prove that the conceptual framework facilitates the process of building UbiApps. We implemented some use cases to validate different aspects of the framework proposed and to testify its capabilities, supported by the elaboration of competency questions, which is a mechanism mostly used to evaluate ontologies, but can also be used as an assessment to this work.

The remainder of this paper is organized as follows. First we compare our approach with related works in Section II, then in Section III we present an overview of the main concepts used as a basis for the framework's development. In Section IV, we describe the framework's overall architecture

[1]www.tempo.uff.br/smartandroid

as well as, in Section V, its most important features. Our framework reference implementation as the assumptions, techniques and features related are discussed in Section VI. A proof of concept demonstrating the feasibility of building UbiApps using the framework is presented in Section VII, showing examples of applications that explores the key features of the IE. Finally, in Section VIII we conclude this paper with the main remarks and future works.

II. Related Work

The benefits to users and developers that UbiComp foresees reach beyond the horizon of many imaginative researchers. The possibilities that arise from it surpass the bounds established by common interface devices, such as mice, keyboards and touch screens. Nonetheless, building such adaptable applications requires much effort from developers who see themselves immersed in a universe of device's specifications and communication technologies, apart from the problem itself that the application must solve. Therefore, many researchers have focused on diminishing those obstacles by providing abstractions, middlewares, services, tools, and other supportive techniques, not only for development but also for prototyping of UbiApps.

The Dey and Abowd framework [3], implemented by the Context Toolkit, is a seminal work for the supporting of rapid prototyping of context aware applications and provides a set of abstractions for the composition of reusable components that focus on context acquisition and interpretation. Another pioneer in the field, the Gaia middleware [4] was designed to facilitate the construction of applications for IE. It consists of a set of core services and a framework for building distributed context aware applications. The Gaia embraces many goals that include the acquisition of context, the maintenance of hardware and software descriptions, mechanisms to find resources and to compose context rules, among others. However, these two approaches do not have concerns on the user side and their preferences, including the prototyping and management of UbiApps at user level.

The framework proposed in DIOS++ [5] offers abstractions to manipulate sensors and actuators, a communication layer to manage distributed sensors and actuators, and a distributed rule engine. But concerns on temporization of context rules and creation of rules at end user level were not touched upon.

The CASS [6] is a simulation rule environment that have a simple mechanism to discover possible conflicts between rules that are composed only using "AND" clause. However, it does not deploy the rules in a running environment.

The DiaSim [7] presents a simulator for pervasive applications, but does not support dynamic configuration of entities in the system. Furthermore the work is not clear about the transparency between real and simulated devices. Other works on simulation of pervasive systems [8], [9] provide a 3D visualization of the environment as in a game. It is possible to configure devices within the system and interact with them. However these works focus on interaction tests of devices and simulation of the physical environment, not worrying about context awareness.

The BASE [10] provides a minimal middleware for handling the resources by means of simple abstractions to access remote services and device-specific capabilities, but it focus only on the communication concern.

The solution presented in this paper strives to accommodate central issues in the development of UbiApps, as identified by the challenges (i), (ii) and (iii), and the cited related works. The conceptual framework we propose facilitates the construction of those applications but don't restrict its design and development options, being flexible enough to accommodate other concerns that are application or support system specific.

III. Overview

It is easy to notice that the context information is a first concern topic in ubiquitous systems. But what is "context"? Many authors have proposed definitions to this concept, though most are little accurate [11]. Synthesizing previous definitions, Dey and Abowd have proposed that context is "any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves" [3]. In a IE "Places" are the rooms, the floors of a building, or spaces in general that enable the localization of other entities, "People" are individuals that populate the environment and interact with it, and "things" are virtual representations of physical objects or software components.

Context aware (or context sensitive) applications is a designation to those applications that not only are capable of knowing the context of the environment, but also react to it either by means of changing the environment or in a software level.

Ubiquitous systems are a class of distributed systems. So, our approach incorporates well established distributed system concepts that are essential to disseminate and manage context information. Besides this, a distributed component model is adopted to implement the entities that live and interact in the ubiquitous environment.

IV. Framework Architecture

The framework provides mechanisms typically required in context-aware applications focused on IE. Among the features is the ability to abstract details that involve communication, context acquisition and resource location, by the applications. For the sake of abstracting resource details we have used components called *Resource Agents* (RA). The RA is the basic modularization unit of the framework being used in the modeling, implementation and management of applications (Subsection V-A).

The general architecture of the framework includes a Distributed Components Model, which defines the RA structure and the Resource Management Support Services (RMSS). The Figure 1 presents a layered architecture that highlights main aspects of the framework. The layers are composed by the following items.

1) Physical Resources Layer: The deepest layer is the one where are found the resources present in the environment. In a IE these resources are the smarter versions of daily used objects, such as beds, stoves, TVs, clocks, and so on. Since all these resources have their own mechanisms of communication

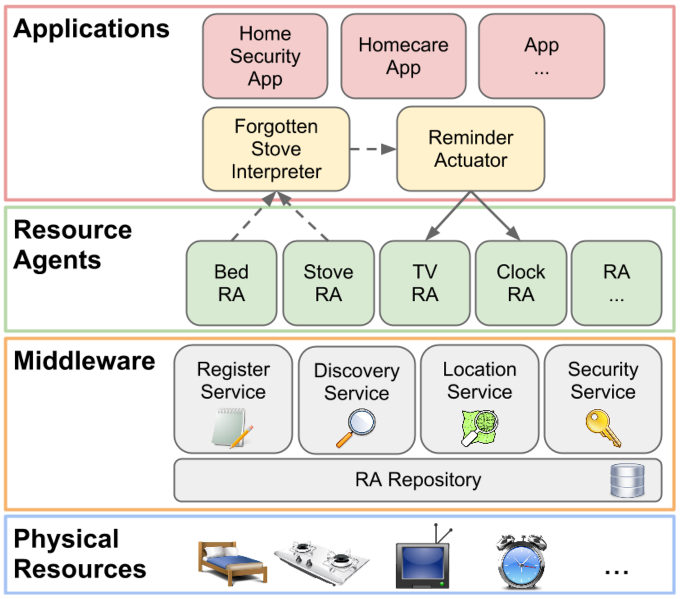

Fig. 1. Framework Layer Architecture

and operation, then a higher level entity is required to deal with them.

2) Middleware Layer: This layer comprehends the main services that sustain the interoperability of the RAs and provides basic features for the development of UbiApps. The management services and the RA Repository, that compose the middleware, will be further discussed in Subsection V-B.

3) Resource Agents Layer: This layer is composed by all the RAs that represent physical resources already living in the IE. The RA will be presented in Subsection V-A.

4) Applications Layer: The higher layer includes all UbiApps that are enjoying the middleware features and also all the software level RAs, which are represented in the figure by the "Forgotten Stove Interpreter" and the "Reminder Actuator".

V. Framework Features

This section describes details of the main features provided by the framework, which includes the definition of the RA (Subsection V-A), the management services (Subsection V-B) and the context interpretation (Subsection V-C). In the next section we will present the current status of the implementation.

A. Resource Agents

A resource may be defined as any hardware or software entity that exposes services (of data acquisition or actuation) which can be used by other entities and applications present in IE. Thus, sensors, actuators and smart devices (e.g. stove, fridge, air-conditioning), as well as software modules that provide a service to the environment, fall within this definition.

The RAs are entities that represent resources. They encapsulate the specificities of resources and expose their information to the environment through standard interfaces, so that others can access them uniformly reducing the complexity of integration.

The Dey and Abowd's definition to context proposes three entities as the most basic of a ubiquitous system, i.e. person, place and object (or thing). Regarding these definitions, we have conceived all these three entities as RAs on the framework, but observed their particular features. Thus they inherit the convenience of being a RA.

1) RA Interfaces: The RA architecture includes two options of interface used for different purposes. The former provides access to Context Variables (CV) and is responsible for exposing the RA's context information. The latter is called Operation (OP) and is responsible for exposing internal attributes (reading) or activating internal attributes (writing) belonging to RAs, enabling applications to manage RA's internal state and explicitly interact with the environment.

Consider the simple example of a smart stove where the instantiation of its RA exposes services to get its current context (e.g., oven is turned on, oven temperature in $^\circ$C) and services that may change its context (e.g., turn oven off, set temperature as $100\,^\circ$C). Hence, the RA concept meets the first challenge (i), which is the *devices heterogeneity*, since decoupling sensor and actuator from the sensor platforms favors the flexibility.

2) RA Structure: In analogy to object-oriented programming we define classes of RAs to represent the type of the resource with specific interfaces. Besides other information present in the AR's structure (such as name, location, subscribed RAs, among others) it has its type hierarchy. The type hierarchy is composed by a sequence of classes that classify the entity and has been inspired by the ontology described by Ranganathan [12]. In the Subsection V-B we will present how this feature can be used to find a RA and to define its functionalities.

3) Communication: In order to meet requirements typical of distributed environments, two communications mechanisms have been included in the framework: a standard Remote Procedure Call (RPC), that implements request-reply communication; and a publish-subscribe mechanism (pub-sub) [13], that implements asynchronous communication. The RPC is used to access directly the RA, querying for its context by accessing its CVs, or calling its internal OPs. On the other hand, through the pub-sub a RA can subscribe to a CV of another one (consumer role) and further receive a notification, when the first RA publishes its CV state change (provider role).

The framework includes a RA Name System (RANS), that allows its unique identification in the IE. This identification is directly mapped to the Internet DNS, thus the names remain the same even if the network address (IP) changes, the only change is the updating of the addressing table. This naming and addressing scheme can be generalized to work over the internet, and comply with the Internet of Things concept.

B. Resource Management Support Service

A set of services is responsible for providing basic functionalities to manage the RAs. The four main services are the RA Register Service (RRS), the RA Discovery Service (RDS), the RA Location service (RLS) and the Security Management Service (SecMS). All these services access the RA Repository (RR) that maintains information about the RAs (addresses and descriptions) and the map data structure (map of the environment). The Figure 1 represents, in the Middleware layer, the RR and above it the management services.

The management services are also designed as RAs, hence they take advantage of the communication mechanisms already provided by it and communicate using the same primitives.

1) RA Register Service: The RRS allows the registration and deregistration of RAs instances references and description. This service also has CVs that allows the developer to know when RAs enter and leave the IE

2) RA Discovery Service: The RDS allows the search for registered RAs based on specific attributes, which are the RA name, RANS and type. The search by RANS will only return one RA, but the search by name or type will return all RAs that meet this criteria.

After the discovery of a RA, the stakeholder can instantiate a stub (i.e. a proxy) of this RA, when both the stub and the current implementation of the RA implement the same interface. Through the stub the CVs and OPs can be accessed, what internally will be converted to network calls to the RA object (RPC functioning).

During the application initialization the only information required to get started is the RDS address. This information can be obtained, if within a LAN, by sending a broadcast message with the request; or using the embedded DNS service, if within a WAN. With the RDS address on hands any RA can be found, including the other management services (i.e., the RRS, RLS and RSS).

3) RA Location Service: The RLS explores the map of the environment to run the required queries. This service performs searches related to the location of RAs, such as the physical location of RAs, the RAs located on some place, the RAs of some type located on some place, the RAs close to another and order by proximity, among others. Thus, if a developer needs audiovisual devices close to some person, for example, she can run this search through the RLS and will receive a list, where an utility function can choose the more relevant result.

The RLS exposes CVs that allows not only to query if a RA is located at a particular place, but also to subscribe to that place for the sake of being notified of the entrance or exit of a specified RA type. Therefore, even if the RA is still not registered, an application can subscribe to a place and wait for its appearance.

Nowadays there are several technologies to map and/or track the location of entities, that may include from people to small things. These technologies vary mainly based on precision, ubiquity (capable of being "invisible") and price. As examples we have the GPS, the smart floor, cameras (associated to image processing softwares), Wi-Fi triangulation, RFID tags, and so on. In this approach we do not comply with any specific technology, and we assume that the RA can discover its own location or the RLS can.

4) Security Management Service: The security concern is always a central topic on UbiComp systems, due to the diverse aspects of security required [14]. Some aspects are already inherited, depending on the technology adopted, e.g., the Wi-Fi already ensures against the registration of outside devices and the exchange of information. Thus our proposal for the the SecMS discusses requirements at a higher level, and issues at a lower level are defined according to the platform used.

In UbiComp, any human-machine interaction should be as inconspicuous as possible. However, on what matters to the users there is a need for authentication what may happens through explicit interaction, such as entering a password, fingerprint, eye retina recognition, and so on. Other less intrusive ways are progressing, such as the facial recognition, but this raises privacy issues. The users fit in user groups that restricts their access to domains in the IE. For instance, in a smart home the father and the mother may have administrative permissions, below them there are the other occupants (such as elderly and older children) that have some level of permissions, next the younger children, and lastly the guests.

A domain can represent a set of applications that share a common characteristic (e.g., home health care, home surveillance, multimedia), or a set of resource within the same constraints (e.g., adult, children, guest). The SecMS allows modifications on security take effect on the level of middleware. Neither a user can manipulate resources or UbiApps that she has not permission to, nor an application can manipulate resources from outside its domain. The specification of domains complies with the theory of sets, thus an entity may be contained in more than one domain, and so on.

C. Context Interpretation

Considering the multitude of resources present in the IE, the context interpretation aims at providing a higher level context information hence allowing the reuse of the logic and a separation of concerns. This feature helps to solve the challenge related to the *variety of context information and services*, what is possible via the RA's interfaces, the CVs and OPs.

The context interpretation serves to aggregate context information from different sources in accordance with some specific logic and considering the passage of time. This logic is defined by the developer and the interpretation is implemented with the framework's support.

The entity that performs the context interpretation may be encapsulated by a RA, thus allowing the subscription to other RAs, what is called Context Interpreter (CI). This concept allows dealing with context in a higher level and also provides a separation on concerns, since a context interpreter decouples the steps for context acquisition, evaluation, timing and notification, that are repeatedly performed by the UbiApps. Therefore, using the CIs also promotes the reuse, either by the creator application or by other ones.

The framework uses a generic approach for creating context rules, which are composed by two parts: the interpretation and the actuation. The interpretation part, as already stated before, concerns on binding the CI itself with CVs from different RAs, evaluating the expression that interrelates those CVs, and notifying whoever is interested. The actuation part is a RA that is subscribed to the CI, and develop some job, that may change the context of the IE or change some state in a software level.

An example of context rules has been presented on Figure 1. At the Applications Layer a set of two components implements the interpretation-actuation pattern: the "Forgotten Stove Interpreter" and the "Reminder Actuator" respectively, where the goal of the rule is to remind the elderly that he has forgotten the oven on while he went to sleep. The former component, which is a CI, is subscribed to two RAs, namely the "Bed RA" and the "Stove RA" (the dashed arrow means notify subscribers). The latter is subscribed to the CI and calls the RAs, namely the "TV RA" and the "Clock RA", directly using RPC (represented by the straight arrow). Internally to the CI a rule expression relates the RAs as "bed is occupied and oven is turned on for 30 min", which means that the CI only will be valid if this condition remains true during "30 min". When the CI changes its state from invalid to valid (what includes the timing), the subscribers are notified, the "Reminder Actuator" in case. This actuator, in its turn, calls the OPs "fire alarm" from "Clock RA" and "set message" from "TV RA", with the parameter "You forgot the oven on!". Then it is expected that an accident is prevented.

An actuator can be simple as a list of actions to be invoked sequentially, such as in the previous example, or it can be more complex, for example, a program that does a mathematical calculation, or that aggregates other communication technology (such as SMS), and many others.

The independence of the actuator allows its binding and unbinding with some CI in a easily manner, what is make possible because of the RA communication mechanism. Based on the same principle, more than one actuator can be bound to the same CI, for example, we may bind the forgotten stove CI with a "family warning actuator", that contacts a relative by phone and put her in touch with the elderly.

Another aspect of context rules is the time dimension. The definition of timers inside a rule expression can be done by subscribing the CI to a Timer RA, hence enabling the direct reference in the expression. The timer can also be relative or absolute. An absolute timer is objective: if the timer has reached some timestamp, or if it is within a window of time, then it is valid.

The solution for implementing the context rules within the framework may not be followed strictly as presented. The framework is flexible enough to permit other forms of context interpretation, such as by aggregating a off-the-shelf rule engine (e.g., Jess, Drools, Clips). A rule engine is a tool generally centralized, which receives context information from various sources and processes rules previously inserted, where it is common to use an algorithm such as RETE [15], processing rules built exclusively with the clause "AND" efficiently. Thus, the developer can use a different approach for rule processing but still enjoying the framework's features for context acquisition, discovery, and so on.

Through the CI the framework address the challenge (ii), which is related to the amount and diversity of context information and services, since the context interpretation allows the autonomous behaviour of RAs.

VI. SmartAndroid

The conceptual framework proposed at this work was tested through a reference implementation called SmartAndroid [2]. The project includes a programming API to help the developers in constructing UbiApps as well as offers basic management services to provide resource registration, discovery and localization. The SmartAndroid has been developed over Google Android platform as we believe that in a near future it can be widely used in sensors and devices embedded in furnitures and home appliances. In addition, this option helped us to implement emulators of several devices using cheap Android cellphones an tablets.

[2]For more information visit www.tempo.uff.br/smartandroid

The SmartAndroid middleware contemplates the possibility of running multiple applications in the same IE. Thus, focusing on the user and inspired by the Google Play – which is a application market for Android devices –, we have as a future work the creation of a SmartAndroid Market [16], where users will be able to easily download and start different applications. Also similar to the Google Play, a Manifest file will require permission for accessing the domains, what will allow users to maintain control over what off-the-shelf applications have permission to know and to do. Within the market, the UbiApp could be tested towards discovering any possible harm that the application could do and any possible inefficiency that could consume too much the energy resources of the host device.

A. Communication Support

To perform any interaction with a RA that is not locally instantiated, the UbiApp must use a stub (also called proxy) that is bound to the referred RA. This stub implements the same interface as the RA, i.e. the same CVs and OPs, which are methods, from a Object-Oriented Programming perspective. The stubs could also be automatically generated since their role is to forward the calls to the implemented RA. With the stub on hands, the application can invoke remote methods, as if they were locally invoked. These remote methods may be to retrieve a CV or to trigger an OP. The Figure 2 represents this direct, or synchronous, communication. The steps to achieve a direct communication, as represented by the figure, are as follows:

1) The UbiApp calls a method from the stub
2) The call is forwarded to the RA instance through the network in the second step
3) This instance can interact with the device (the lamp, in case)
4) The result or the acknowledge is then returned to the caller

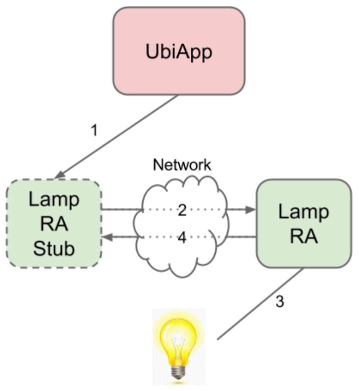

Fig. 2. Direct communication

It is possible to acquire the status of a CV directly, however it can also be done indirectly via the pub-sub mechanism, what makes it easier for developers being aware of CV change right away. This indirect, or asynchronous, communication is performed in two stages: the subscription (Figure 3.a) and the notification (Figure 3.b). During the first stage, represented by the Figure 3.a, we have the following steps:

1) The UbiApp sets some RA ("RA X") to subscribe to another, referred by its RANS ("Lamp RA") and in a specific CV ("Is on")
2) Then the first RA requires to be subscribed to the second, through the network
3) The second RA accepts the subscription and is bound to the device

The second stage happens when the first RA is already subscribed to the second and a notification routine is triggered, what was represented by the Figure 3.b and works as follows:

1) The device has its context changed, what may be caused directly (e.g., a user turning the lamp on) or through the RA (i.e. an application, another RA, or rule, calling some RA's OP which caused the change of context)
2) Then the RA that represents the device notifies all subscribers to that CV (e.g., "Is on" from this Lamp RA)
3) The subscriber may call some callback method of the UbiApp

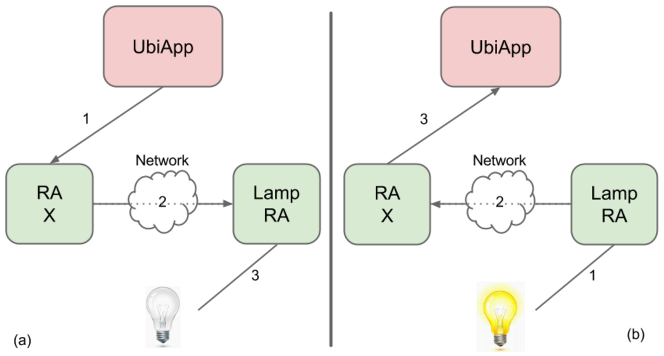

Fig. 3. Indirect communication

We have provided a simple implementation for the pub-sub mechanism, which was effective to evaluate our approach. The Figure 4 shows how we have implemented this mechanism using the RPC. In the example, two RAs (TV and Alarm Clock) subscribe to the Stove RA (step 1). Every RA keeps track of its stakeholders, where the stakeholder list keeps the RANS and the CV of interest. When a remote control (or another actuator) calls a OP that changes the state of the Stove RA (step 2), this is intercepted (step 3) and causes the selection of the stakeholders (step 4) followed by the notification (step 5) that also uses the RPC.

B. Context Rules Composition

The SmartAndroid supports the context interpretation by providing mechanisms to create context rules. The composition of context rules can be done basically in two ways. The former

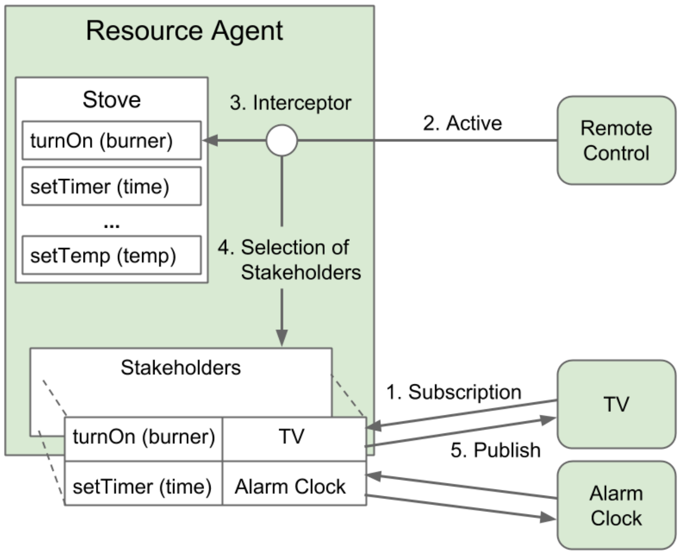

Fig. 4. Resource Agent

is at the implementation level, where developers are able define their context rules in a description file or using a programming language. The latter is at end user level, where non trained people are able to define their context rules supported by the PIPA (Subsection VI-C).

The context rule life cycle has two major phases: composition and running. The composition phase corresponds to the instantiation of a CI and of an actuator and the binding of both through subscription. The running phase is when the CI and the actuator(s) are already instantiated as the other RAs that feed the IE with context information, and they are able to exchange messages and perform their job.

At the implementation level, a context rule can be described in a JSON structured file (or byte stream) following a specific format, thus the CI code and the binding with the actuators can be automatically generated. The file must contain the IC definition with the rule expression, and may contain the definition of actuators too. This expression contains conditions that are the comparison of RA's CVs with constant values (e.g., Stove.OvenTemperature = $180\,^{\circ}\mathrm{C}$), of two different CVs from the same or from different RAs (e.g., ArConditioning.Temperature > OutsideSensor.Temperature), or the RA's CV alone if it is has a boolean value (e.g., Stove.IsOn). The conditions are bound to each other through the logical operators "AND" and "OR", it is also allowed the use of "NOT" and of parenthesis, what makes possible to compose more complex expressions by using subexpressions. The expression or subexpression may also contain a timer.

The definition of actuators is simpler than the definition of the CI and is not mandatory, since other actuators can be bound to the IC later. A actuator is composed by a list of actions, where each action have a RANS, a OP reference and the parameters (e.g., Stove.SetOvenTemperature to $100\,^{\circ}\mathrm{C}$). Both the actuator and the CI have a name, that makes easier to search for them.

The SmartAndroid has a parser that can read the JSON file and create the instances of the CI and actuators. After reading the rule expression the parser creates a tree data structure that corresponds to that expression in memory, and the timers are also referenced within the structure. The CI created queries all the CVs that appear on the expression to keep a cache of the context information and to start the evaluation, next the CI subscribes to all these CVs and sets itself as “running”. After the creation of the CI, the parser creates instances for the actuators, which are subscribed to the CI previously instantiated.

A common concern when having multiple context rules running in parallel is the occurrence of conflicts. The conflict happens when two CIs may be validated at the same time and the actuators associated with them have opposite actions. The conflict has also happen when a context rule causes the triggering of another that may cause the triggering of the first one, leading to a circular deadlock. We have as a future work to focus on this concern and to provide a solution for the detection and resolution of conflicts.

C. Interface for Management and Prototyping

The main purpose of the PIPA is provide to developers a hybrid environment where simulated and real resources can interact to each other transparently, enabling their users to construct UbiApps as well as test and evaluate them through a wide variety of scenarios. Thus RAs from sensors, actuators and household appliances can be graphically represented at our tool, along with a schematic map of the IE so that some of the resources aspects, as its context and location, can be easily visualized on the map, helping in application debugging. The resource are represented as icons that can be moved around the map in order to simulate the intended application behaviour. For example, an alarm message can be displayed on the tv screen placed were the person is seated watching the TV.

Besides, PIPA operate with two different interfaces: the *developer interface*, already mentioned, and the *end-user interface*. In the end-user interface, users in general, i.e. non-technical users, are able to use PIPA to manage their IE remotely using a mobile device, such as a tablet. Through an intuitive GUI, end-users can not only visualize the state of their home devices but control them. For instance, it’s possible turning off a stove burner, change the air-conditioner temperature, and so on. By touching the device icon in the screen, an interface that presents the device’s main functions is shown, where the user can operate it as the tool were a remote control.

In addition, end-users can compose context rules in a high level way by graphically combining the CVs from any RA and logical connectives, forming logical expressions as *“IF the TV from living room is on, AND nobody has been present at the living room for 15 minutes, THEN turn off that TV”*. To compose this rule the user have to, first of all, select the rule composer mode in the PIPA. Then, he must touch the TV icon at the living room representation on the map, and choose the desired CV – in that case, the CV that refers if the TV is turned on or not. After that, a dialog box showing all possible logical operators is shown, so that the user can select one of them as the “AND” operator. The user can repeat this process, adding others CVs and operators until the expression is complete. Finally, the rule can be associated to one or more actions, provided by actuators. When the context rule is finished, the

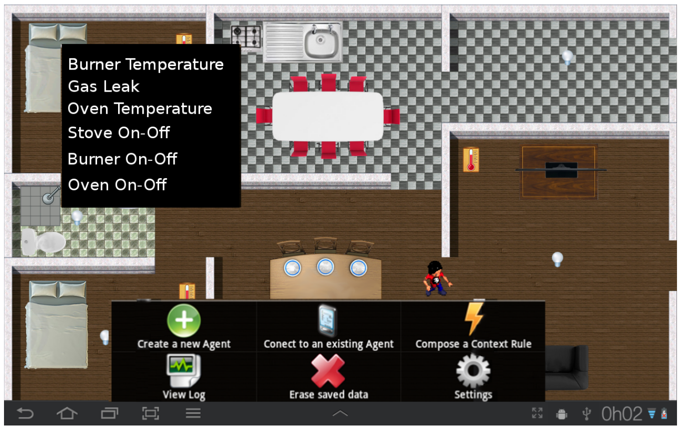

Fig. 5. Management and Prototyping Interface

SmartAndroid writes down this rule in a JSON file respecting the standard, thus if the system crashes or terminates, all the composed rules can be afterwards re-started.

VII. Example Applications

Some applications have been built to explore the potential of the framework for building UbiApps. Although the example applications are simple, and alone does not justify the deploy of the SmartAndroid in the environment, they have allowed us to validate the features provided by the framework and achieved by the SmartAndroid. It also demonstrated the possibility of concomitant running of UbiApps, what similar to an operation system where different processes run separately but in the same system. Within the SmartAndroid, UbiApps run separately but in the IE.

In the subsections that follows we will elaborate over some UbiApps that have been developed, or are still in development, in the course of the project. Besides these applications we have also implemented some context rules, such as the ‘Forgotten Stove” that have appeared throughout the text.

A. Tic-Tac-Toe

After the implementations of the basic features of the framework, including the communication mechanisms, we have implemented a distributed Tic-Tac-Too game, where a player can defy another player (or AI player) placed anywhere in the IE. We have used as devices two smartphones.

In this implementation we have created RAs for both the players and the board. The players subscribe to the board and vice-versa, thus each movement performed by the player is set on the board through RPC, commands from this player are locked and the movement is communicated to the other player, the same happens to the second player, and so on. One of the goals of this implementation was to verify the capability of the framework to make the construction of distributed systems easier.

B. SmartLiC

The Smart Light Control (SmartLic) application aims at the clever use of lighting in homes in order to save power consumption, since it is responsible for the largest range of consumption in residential environments. The application uses presence sensors to identify the location of people inside each room, and decide which lamps should be turned on and off. Hence it prevents the forgetting of lamps turned on, and may lit some lamp if a user enters a room at night and if nobody is at sleep in that room.

The PIPA has allowed us to test this application. We instantiated lamp simulators for each room and an avatar for a user, and also a simulator for day light sensor. Next context rules were created for each room of the house. The following enumeration shows the initialization routine of the SmartLiC UbiApp.

1) Create a context rule skeleton in JSON
2) Get the rooms from the house
3) Get the lamp and presence sensor of each room and fill the rule expression skeleton
4) Instantiate a CI with the rule expression
5) Instantiate a actuator to perform the actions declared in the context rule skeleton
6) Bind the CI and the actuator
7) Repeat this for each room

The event of a user entering some room is triggered when the avatar is dragged from one room and dropped on another. The CI has a timer associated with the expression so, when the rule is valid, the timer is fired. If the context change and the rule is invalidated the timer is stopped, otherwise the evaluation is completed with a valid rule and the subscribers must be notified. Then the actuator turns off the light of the first room.

There were also rules composed for turning on the light of the room that the user enters if it is day (based on a day light sensor simulator) and if nobody is at sleep in the room (based on another CI).

Another feature that we will implement is a house security mode, that users may activate if they are travelling or just have left the house alone. The basic operation is to turn lamps on and off as if users were at home, thus deceiving potential thieves. The choices of which lamps to turn on and off can follow real sampled patterns, stored in a log file, to mimic realistic living scenarios.

The SmartLiC allowed us to validate two features of the RLS, which are the discovery of the rooms and the RAs of some specific type are within the room (i.e. the lamps and presence sensors). In this UbiApp we also have created a set of context rules with timers using the context rules parser (including CI and actuators). And the PIPA were well explored in this implementation, since we has shown the triggering of many events in parallel and multiple users cohabiting the same IE.

C. Remote Controls

Considering that the users would want to have some control over the resources within the IE, we have implemented the remote controls to interact with them specifically. The same way that users nowadays have a TV control that enable them to have full control over the device with little effort, using the framework that task will be easily extended to the diverse range of devices present in the IE.

For implementing a lamp remote control, as shown on the Figure 6, we are using a prototyping board called BeagleBone [17]. The board, on one side, is capable of turning a lamp on and off and, on the other side, it implements a RA for the lamp which is registered in the IE. A graphical interface of a lamp can remotely call these OPs of the lamp, and it is also subscribed to the Lamp RA, what allows the remote control being aware of the lamp status.

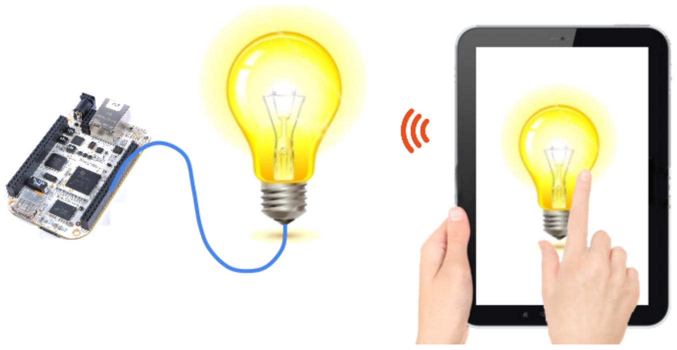

Fig. 6. Remote Control

D. Prenda Game

The Prenda is a ubiquitous game implemented for the SmartAndroid where the goal is to find a gift hidden somewhere in the house. The player moves inside the house and gets tips on where to find the gift while competing with others, the one who finds the gift first is the winner. This game has a different concept of other common digital games, since it explores elements in the IE and thus offers a different experience for kids.

This game was implemented with simulated resources and using the PIPA. The players were represented with avatars and their location were updated with the drag and drop of the avatars over the map.

The player has a layout map of the house where you can find the move to attach the extent that the movement occurs, the game informs the person's subjective closeness with the gift (by the number of points in the message text) to that is found. The game allows the participation of multiple players for the environment, to get a winner the gift is repositioned to the new game. The game takes place in a simulated environment, using IPGAP to simulate the movement of avatars of people participating.

E. MediaFollowMe

The MediaFollowMe application enables the user to be followed by any kind of media stream (e.g. video, audio, images) while he walks inside the IE. By the time the user launches the MediaFollowMe application and choose the device in which the media stream he wants to be followed, the application will take care of the user's location changes and automatically play, pause or migrate the media to the closest device capable of displaying it.

To illustrate that, consider a user watching a movie on a TV through a blu-ray player in his bedroom. When the user leaves his bedroom, the system identifies this action by an

RFID tag and fires an event, as well as when he arrives in any room. As the UbiApp is interested in these events, it receives a notification informing that the user has left/arrived some room (and which room) and can migrate the media to the proper device, preferably inside that room.

VIII. Conclusion

In the present paper we have proposed a framework that address some major issues that are found in the construction of ubiquitous applications, both for developers as for end users. We have focused on three challenges, that have guided the development of the framework. For dealing with the heterogeneity of devices present in the intelligent environment we have created the Resource Agent abstraction that encapsulates the mechanisms to interact with physical and also virtual resources, and includes synchronous and asynchronous communication mechanisms. The variety of context information and services is achieved by the context variable feature, which allows the creation of customized rules. These rules define Context Interpreters, that are responsible for aggregating context information, providing it in a higher level. And for the resource availability issue we have created the PIPA, which is a software that helps the prototyping and testing of ubiquitous applications through a easy to use interface, and allows the visualizations of applications running in a environment populated with real and simulated devices.

The concepts of the framework were implemented as a middleware called SmartAndroid. The SmartAndroid provides a API for developers and a set of basic services, that enables the construction of the intelligent environment. The core management services are the resource register service, that includes a naming service; the resource discovery service, that performs searches on registered resources based on different criteria; the resource location service, that also performs searches but based on the location and proximity of resources and/or users; and the security management service, that allows the customization and ensures the limits of users and applications within the environment.

As future work we plan to extend the framework to allow more complex queries in the resource discovery service, what nowadays is still strict; adapt the communication mechanism for using web services, and get our approach closer to the concept of the internet of things; further research on other security concerns; provide mechanisms to avoid/solve potential conflicts on context rules; aggregate machine learning techniques that will enable a better understanding of user preferences, and will make possible the offering of context rules and other helping services; improve the PIPA and evaluate its usability, using methodologies of human-machine interface; and son on.

Acknowledgment

The authors would like to thank CNPq (Federal Research Funding Agency) and FAPERJ (Rio de Janeiro State Research Funding Agency) for partial funding of this work.

References

[1] M. Weiser, "The Computer for the 21st Century," *Scientific American*, vol. 3, pp. 94–104, 1991. [Online]. Available: http://wiki.daimi.au.dk/pca/_files/weiser-orig.pdf

[2] R. de Araujo, "Computação ubíqua: Princípios, tecnologias e desafios," in *XXI Simpósio Brasileiro de Redes de Computadores*, vol. 8, 2003, pp. 11–13.

[3] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications," *Human-computer interaction*, vol. 16, no. 2, pp. 97–166, 2001.

[4] A. Ranganathan and R. Campbell, "A middleware for context-aware agents in ubiquitous computing environments," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Springer-Verlag New York, Inc., 2003, pp. 143–161.

[5] H. Liu and M. Parashar, "Dios++: A framework for rule-based autonomic management of distributed scientific applications," *Euro-Par 2003 Parallel Processing*, pp. 66–73, 2003.

[6] J. Park, M. Moon, S. Hwang, and K. Yeom, "CASS: A Context-Aware Simulation System for Smart Home," in *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*. IEEE, 2007, pp. 461–467. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4296972

[7] J. Bruneau and C. Consel, "Diasim: a simulator for pervasive computing applications," *Software: Practice and Experience*, 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.2130

[8] J. J. Barton and V. Vijayaraghavan, "Ubiwise, a ubiquitous wireless infrastructure simulation environment," Hewlett-Packard Laboratories, Palo Alto, Tech. Rep. HPL-2002-303, 2002.

[9] H. Nishikawa, S. Yamamoto, M. Tamai, K. Nishigaki, T. Kitani, N. Shibata, K. Yasumoto, and M. Ito, "UbiREAL: Realistic Smartspace Simulator for Systematic Testing," in *UbiComp 2006: Ubiquitous Computing*, ser. Lecture Notes in Computer Science, P. Dourish and A. Friday, Eds. Springer Berlin / Heidelberg, 2006, vol. 4206, pp. 459–476. [Online]. Available: http://dx.doi.org/10.1007/11853565_27

[10] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, "Base-a micro-broker-based middleware for pervasive computing," in *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*. IEEE, 2003, pp. 443–451.

[11] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[12] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. Campbell, and M. Mickunas, "Olympus: A high-level programming model for pervasive computing environments," in *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*. IEEE, 2005, pp. 7–16.

[13] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[14] T. H.-J. Kim, L. Bauer, J. Newsome, A. Perrig, and J. Walker, "Challenges in access right assignment for secure home networks," *Proc. HotSec 2010*, 2010.

[15] C. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial intelligence*, vol. 19, no. 1, pp. 17–37, 1982.

[16] Google, "Android Open Distribution," http://developer.android.com/distribute/open.html, 2013.

[17] BeagleBoard, "Open Hardware Physical Computing on ARM and Linux," http://beagleboard.org/, 2013.