An Interplatform Service-Oriented Middleware for the Smart Home

Ehsan Ullah Warriach, Eirini Kaldeli, Alexander Lazovik and Marco Aiello
Department of Mathematics and Computer Science
University of Groningen
Groningen, The Netherlands
Email: e.u.warriach,e.kaldeli,a.lazovike,m.aiello@rug.nl

Abstract

In domotic scenarios, where ubiquitous computing pervades housing, the capability to effectually discover, integrate and coordinate different devices with diverse implementation details, communication protocols, and functionalities is a central aspect. Basing on a layered domotic architecture for smart embedded devices which builds on the paradigm of Service Oriented Computing, in this paper we focus on the pervasive layer and the requirements it has to fulfill, so as to realize a high degree of automation and serve the needs of the higher application levels of the architecture. We show how a discovery framework can take care of new devices independently of their protocols through the use of proxies. Because with the development of home network and service applications, different protocols and transmission modes are proposed. There are more digital devices and home appliances which compliance to the protocols. The proposed protocols are different and they are typically unable to interconnect with each other. We design and implement a middleware framework for smart homes to integrate protocols which are popular such as UPnP on OSGi framework and collaborating Zigbee and Bluetooth to converge various service oriented applications. Additionally, with the well-developed Zigbee and Bluetooth technology, majority of devices has been developed to support these technologies, we propose two new base drivers to integrate diverse devices communication protocol on our platform. The availability of standardized service type descriptions can further assist towards automating the tasks of integration and coordination through intelligent algorithms. From the technical point of view, the implementation builds on the OSGi/UPnP standards. An experimental setup for testing the discovery plugins for Bluetooth and ZigBee devices, along with an evaluation of the performance of the Bluetooth proxy verify that the proposed solution is a viable and effective one.

1 Introduction

Smart homes constitute an environment where appliances inter-work with each other and coordinate to proactively support the user in his/her daily activities and in case of special home events. Home appliances, sensors and actuators are highly heterogeneous in a number of ways. They come from many vendors, have hardware and software resources that go from minimal to being as powerful as a desktop computer, and have the ability to change the physical space around them or just to sense it. This high heterogeneity needs to be reconciled to lay the foundations for a smart home, where devices are aware of each other existence and interaction interface. The reconciliation goes through the use of common ontologies, standardized protocols and unified techniques for discovering and messaging.

An extreme solution to the coordination problem is to force all devices to adhere to a unique networking protocol. Even though standardization efforts and industrial alliances have done a great job in the last years to reduce the number of protocols and to come up with excellent standards (e.g., Konnex, UPnP, Bluetooth, WiFi, ZigBee) as well as software pervasive architectures (e.g., Jini), we are still far from the point where all devices follow a common protocol. Therefore, the association and interoperation between heterogeneous home appliances is still an open problem in the context of creating smart homes. In this paper, we focus, in particular, on the problem of discovery and interoperation across different protocols to build a middleware for the smart home.

The functionalities or *services* offered by each device, e.g., turning on/off or setting the channel of a TV, are made available through some specific interface exposed by the device. The problem of service discovery has received growing attention in the research arena. Proposals provide mechanisms for searching and browsing services, for choosing the right service, and for utilizing it. Yet, each protocol has its own communication mechanism, characteristics, and base on different techniques to service discovery, thus leaving many issues related to cross-protocol interoperation open.

In this article, we propose a middleware for smart homes where heterogeneous devices can dynamically connect, and services are offered across protocols. The middleware is the basis for building smart homes in the context of a European Framework 7 project, *Smart Homes for All* (SM4ALL), which has the goal of making homes pro-active to address the needs of people with specific physical disabilities. Here we overview the project and dig into the details of the middleware component named *Pervasive Discovery Framework (PDF)*. This framework is responsible for the physical discovery of devices using low-level communication technologies. PDF is composed of discovery modules, one per each communication technology supported by the framework, thus being able to perform discovery of heterogeneous devices. This discovery framework transforms the devices proprietary representation into a common representation that can be understood by the rest of the middleware layers.

This abstraction from the pervasive layer through the availability of machine-parsable and standardized device descriptions becomes an essential element if we want to move to genuinely intelligent houses, which are able of building efficient and flexible added-value applications, decoupled from the particularities of the functionality implementation, operational platforms and communication protocols used by the underlying devices. One distinguishing feature of the SM4ALL framework is that it offers extensive support for building complex services through the combination of automatic and semi-automatic compositions algorithms [7,14]. These compositions are context-aware and self-adaptable, so as to be able to dynamically adjust to changes in the environment and recover from failures, without sacrificing performance, user friendliness and reusability. To enable the development of such complex and intelligent functionalities that are based on service compositions. To this end, a semantic repository is used to collect and store standardized descriptions of the functionalities offered by each device type, specified in the UPnP standard. This way, every time a new device of a supported type is discovered in the home network, all steps of its integration up to the highest levels of abstractions can take place in a purely automated fashion. The overall layered architecture thus provides for semi-automatic device discovery and integration of the respective services into a centralized gateway, offering uniform management of services to higher-level applications, while keeping the need for human intervention at all these stages to the minimum.

A discussion of related work is presented in Sections 2. We introduce the *Smart Homes for All* project and its global Service-Oriented Architecture in Section 3. A description of a possible scenario in a smart home is presented in Section 3.1. Section 4 is dedicated to the device discovery framework. The device layer and implementation of discovery plugins for each supported technology is explained in Section 5. Section 6 describes the pervasive controller and explains the patterns and protocols of the implementation. Middleware clients, with emphasis to the composition engine, are presented in Section 7. Section 8 illustrates an implementation based example on the concepts deployed in a physical home and the results performance and conclusions is presented in Section 9.

2 Related Work

The advantages offered by Service-Oriented Architectures (SOA), such as platform independence, loose coupling and interoperability are important to build applications that involve different devices in home networks [20]. During the last two decades, several SOAs targeting pervasive environment have been developed, e.g., UPnP [25] or Jini [9]. As a networking architecture for the home devices UPnP [25] is used for over a decade [15] to simplify the implementation of home networks and support connectivity. We therefore choose UPnP as the basic networking technology, without however restricting the pervasive layer to only UPnP devices. Other technologies such as Bluetooth and ZigBee are also supported through the use of appropriate drivers, one per supported protocol.

The development of the OSGi platform [9] has contributed towards these requirements, constituting the most mature technology for bringing the SOA paradigm into the home network of appliances. OSGi is a platform- and device-independent framework that can be used to develop service gateways [17, 18], The OSGi platform offers the means to manage all services and computation tasks at the service gateway in a centralized manner, and is therefore used as a fundamental component at the pervasive layer in the architecture we propose herein. As already mentioned, in this paper we propose to describe all devices as UPnP devices, a protocol that is easily integrated as an OSGi bundle into the common registry. Several approaches propose extending the OSGi framework so as to integrate devices that use alternative protocols, such as SIP (Session Initiation Protocol) [6, 8], so that they can talk to other devices. Their solution can be used to support SIP devices in the common OSGi registry, since the higher application levels (itnerface, composition, visualization) of the proposed SM4ALL architecture do not make a distinction between OSGi bundles, as long as a description of the functionalities of the services exposed at the service gateway exists.

Several proposals for extending the OSGi description with appropriate semantics have been made in the literature, with the purpose to facilitate the discovery process [11] or for supporting their automatic parsing by software agents. Herein, we use established standards (proposed by the OSGi alliance) as they are: assuming that a common data model is used so as to avoid message-level heterogeneity, no extra expressive power or semantics are needed for automatic discovery and composition.

A number of research and industrial projects focusing on supporting a wide range of household devices over heterogeneous network environments have been performed and are underway. The majority of the solutions examines the integration of technology and services through home networking for a better quality of life. Microsoft's EasyLiving [5] is a middleware for building intelligent environments based on XML messaging, integrating geometric knowledge of people, devices and places. The adaptive house [19] allows the home to program itself by observing the lifestyle of inhabitants and then learning to predict their needs, by means of neural networks. The Gator Tech Smart House [12] develops and deploys extensible smart house technologies, employing a service-oriented OSGi framework that facilitates service composition. The distinctive contributions of the SM4ALL middleware proposed herein include the semi-automatic discovery of devices with heterogeneous network protocols, and their integration into a common registry as services, so that they expose their functionalities in a standardized and unified way, and can thus be used by clients in complex applications without the need to know about low-level particularities of each device. This is achieved by the use of a semantic registry, so that every time a new device of a supported type is discovered in the home network, all steps of its integration up to the highest levels of abstractions take place in a purely automated fashion.

Eventing is the basic communication mechanism used in the middleware architecture proposed in this work, with clients-subscribers expressing their interest in particular kinds of events, and devices-publishers which deliver appropriate notifications each time their state changes. An event-based distributed middleware architecture is proposed in [22], focusing on providing high scalability, while the BOSS [24] project deals basically with smaller-scale home environments. The latter makes use of UPnP, but focuses on sensors, which are just event sources and not services that may provide functionalities to manipulate the environment. The middleware architecture proposed in this paper builds on using WS-Notification as an event-based mechanism.

3 SM4All

Smart Homes for All (SM4All) is a European Union research project in the context of the seventh framework program (FP7). The goals of SM4All include the study and development of an innovative middleware platform where smart embedded services can interwork in immersive and person-centric environments through the use of composability and semantic techniques, in order to guarantee dynamicity, dependability and scalability, while preserving the privacy and security of the platform and its users. This is applied to the challenging scenario of homes in presence of users with different abilities and needs (e.g., young, aged and disabled).

For SM4All's middleware architecture we follow a layered approach. It is based on three layers as shown in the Figure 1. On top is the *user layer* which provides the interfaces to the home such as touch screen and a Brain-Computer Interface for disabled people [3]. The *composition layer* receives high-level goals issued by users through the user interface layer and computes the corresponding complex tasks by controlling the execution of lower-level services offered by devices deployed within the SM4ALL architecture. The composition layer details can be found in [14]. At the bottom is the *pervasive layer*, where the heterogeneous actuators, sensors and mobile devices of the house live. It is responsible for enabling physical devices to interact with the upper layer components, by publishing available service descriptors, executing service instances, decoupling interactions from the specific communication mediums, etc. This layer realizes an abstraction layer wrapping all the devices currently connected into the house.

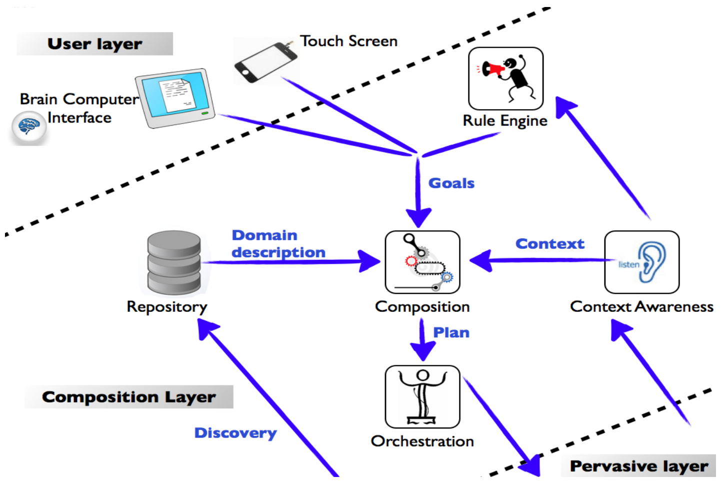


Figure 1: SM4All Architecture

3.1 An interoperation scenario

When a new device enters into a smart home, it first needs to negotiate its integration with the environment. Some fundamental questions need to be answered: Where does it sit in the network? What services does it provide? Does it need to be told what to do by the system? How can it co-operate with the other devices available in the house? To address these issues in an effective way, it is desirable to design a system that can seamlessly integrate new devices without user intervention. Ideally, one would want to take a device to home, and do nothing else. Clearly, zero-configuration is an idealized view, however, in SM4All we try to automate the integration process as much as possible by minimizing the amount of setting up and configuring new devices. The average home user is keen at maximum to provide some input on what are the goals he/she desires to frequently accomplish with the devices available in the home, however, he/she should stay as decoupled as possible from the technical details of the particular implementations.

Let's assume that a house owner, Andrea, wants to integrate a new device into the smart home. Andrea goes to a store and purchases a new device, e.g., a TV. The next step is to establish permanent connectivity between the TV and the smart home platform over a secure network. So, when Andrea comes home with the new TV and installs it by connecting its cable to the network, the TV will start sending basic information to the *home gateway* depending on its communication protocol. The Home Gateway detects the presence of the new device, identifies its type and basic characteristics, and downloads the corresponding control software and appropriate description from a semantic repository (a local or a remote one, e.g., by connecting to the Internet). The *home gateway* exposes the TV services to the rest of the system, and informs the user interface that gives access to all home devices about the operations that control the new TV. After successful installation, Andrea can start using his new device, and the system components can communicate with it.

Let us now see how a newly installed device can cooperate with the other devices in the house. Obviously, the addition of a new device broadens the potentials for buinding added-value, more complex funcitonalities that leverage the degree of automation of the inhabitants' routine. Exploiting the benefits offered by a newly discovered service and properly incorporating it into existing composite services should take place with the minimum amount of effort, without the need of re-designing the whole composite service and independently of whether it uses a different protocol than the other devices which participate in it. For example, let us consider a "good night" goal which is a request addressing to the composition module (see Section 7.2 about how the goal is specified in a declarative way), and prescribes the desired bedroom conditions that the user wants to accomplish when going to sleep: the alarm clock is set to some preferred wake-up time, the lights in the bedroom are turned off, and the curtains are closed. Let us assume that a disabled user of the house buys a motorized bed, which after being plagged is automatically detected by the system, and added to the homeway, after deriving its description from the repository. Let us also assume that this description derived from the repository also includes a set of semantic descriptions (see Section 7.2 about how these semantic descriptions look like), which enable the automatic composition engine to use the device in is complex reasoning tasks. All the user needs to do in order to take into account the functionalities exposed by this new acquirement of his in his going to bed routine, is to add an extra requirement about the bed to be at the low position (see Section 7.2). Thanks to the automatic recognition of the type of the bed device and the retrieval of its semantic description, which is consistent with the description of the rest of the devices and can be automatically parsed by the composition engine, nothing more is needed: the next time the user wants to sleep, on top of the other service invocations that will take place, the motorized bed will also be automatically lowered, despite the fact that it uses a different protocol from the other devices which are invoked by the plan.

4 The Framework

The pervasive framework is a dynamic and open environment where devices join and leave while offering and consuming services. It is an extension of framework shown in the Figure 2 by the addition of the Semantic Repository component. In order to support scenarios such as the one described, the smart home platform has to satisfy a number of requirements. Firstly, a new device should be automatically detected and installed on the *home gateway*, when joining the physical network. Secondly, all interested parties components of the platform should be notified about the services offered by the newly detected device. Thirdly, the available services should be described in a standardized programmatic manner, so that the subscribed clients-components can control them in accordance with this description. Fourthly, interested parties should be notified about changes of services' states in a event-driven manner, and communication between services should be enabled regardless of the platform each service runs on. On top of these, the pervasive layer and discovery framework have to be scalable and secure, and perform well with varying loads and number of participating devices.

The pervasive layer of the SM4ALL platform relies on a service-oriented and event-driven architecture. Figure 2 illustrates the current implementation of the pervasive framework. At the bottom is the device layer, where the physical devices are located (see Section 5). UPnP devices use TCP/IP and UDP as basic networking protocols to communicate with the pervasive controller. Other heterogeneous devices, use their own standard communication protocol, such as *Bluetooth*[1] and *ZigBee*[2] through the deployment of discovery drivers. The *pervasive controller* is a special OSGi bundle responsible for handling eventing and control of the UPnP services available at the OSGi framework. It thus functions as a bridge between the OSGi layer and the Web Services (WS) layer, which provides a standardized API to the upper layers (more details about the pervasive controller are provided in Section 6).

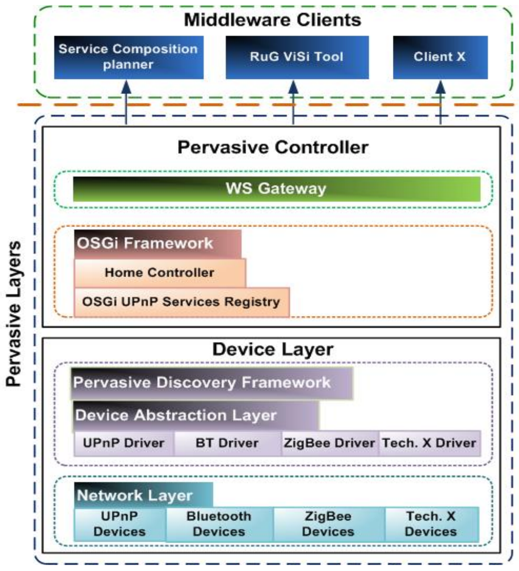


Figure 2: Architecture of the Pervasive Framework

On top of Figure 2, there is the *middleware client layer* which contains middleware clients, which communicate with the pervasive controller, and can invoke the services exposed by the available devices. The clients may include a visualization and simulation tool (see Section 7), a BPEL orchestration engine [16], or a composition layer [14], user interface [7] etc.

4.1 Protocols

We use Universal Plug and Play (UPnP) [25] as the protocol for direct access to hardware services, WSDL and SOAP protocols to expose high-level services, and the Open Service Gateway initiative (OSGi) framework is as middleware technology.

4.1.1 Universal Plug and Play

The UPnP protocol plays the role of a device-neutral technology which is used by the device abstraction layer, required to fulfill the requirement for device independence. According to the UPnP specification, a device includes a set of services, each of which maintains some actions, i.e., operations that can be invoked, and involves some state variables, which model the current state of the service. UPnP archives platform independence because it is built upon standard technologies such as TCP/IP, UDP, HTTP, XML and SOAP. It offers support for discovery, description, control, event notification and presentation [25]. Three kind

[1]http://www.bluetooth.com
[2]http://www.zigbee.org

of components are involved in the UPnP architecture, such as service, device and control points.

In UPnP, a service contains some parameters used to maintain the operating status of the service. UPnP uses an XML based format to describe services. There are two types of elements that can be described using this format: actions and state variables. Actions are used to control UPnP services. State variables describe the current state of UPnP service and they may send events when their state changes. UPnP services can be controlled by invoking their actions. These invocations go via SOAP request/response messages. UPnP services support eventing using General Event Notification Architecture (GENA) as shown in the Figure 4. GENA uses HTTP as a request with an XML message body.

4.1.2 OSGi

The OSGi framework is a dynamic module system and service platform for Java [21], whose original purpose was to build service gateways in a server-centric architecture [28]. It provides a central coordinating point for managing the home network with multiple heterogeneous communication technologies, and allows services from different devices to be loaded and run on a common service gateway. The framework is meant to facilitate the deployment of services and applications on a residential gateway.

5 Device Layer

The main goal of the device layer is to seamlessly integrate heterogeneous networks and devices (sensors and actuators) into the pervasive discovery framework and provide devices services and information using a common and standard device abstraction interface, no matter which underlaying technology is the device based on, as shown in Figure 2.

5.1 Discovery Framework

A dynamic, adaptable discovery process is needed to find new devices and register their basic information, given that devices are highly mobile, corresponding to physical things that move inside the smart home. Furthermore, there is a need to work in a unified way with devices using different protocols and networking technologies (e.g., Bluetooth, ZigBee, UPnP). In order to enable the loose coupling of devices, a service discovery framework is proposed, which abstracts away the underlying device technology to a common representation, and permits users to find and use services offered by heterogeneous devices without any previous knowledge of their location or characteristics. Moreover, it allows other components connected to the *home gateway* to get information about the existence of a device service and its description. Figure 3 provides a high-level overview of the working of the discovery framework.

The requirement for automatic recognition and integration of devices, independently of their network technology, implies the use of a device abstraction layer. This brings along the notion of abstract device *models*, as opposed to the instance of a particular device and its respective implemented operations. For example, in a home there may be a number of lamps which are instances of the same lamp model. The basic assumption is the existence of a *semantic repository* which keeps the descriptions of the device models the platform supports,

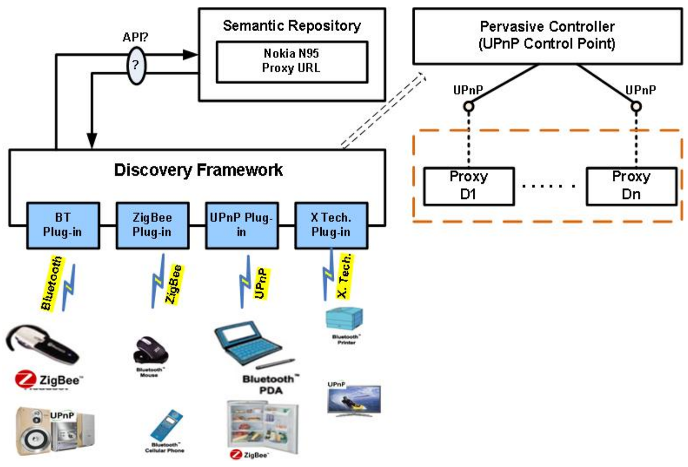


Figure 3: Pervasive Discovery Framework Architecture

specified in UPnP, which is the unified standard we use for the device descriptions. This can be also used for service composition. This can be also used for service composition. This can be also used for service composition [14]

The general sequence of actions that takes place in the middleware framework and particularly in the discovery framework from the moment in which a physical device appears in the network to the one in which its services are available to be used as UPnP for the upper layers of the middleware are:

1. The *home gateway*, which contains the communication technology adapters and one driver listener for each communication technology, such as Bluetooth, ZigBee, etc., tracks periodically the local network for new devices.
2. When the framework receives a signal from a new device, irrespectively of the used network technology, it parses the information conveyed by the basic notification, which includes a device instance unique identifier, and its respective type-model identifier. This process is called physical device discovery. The control flow of the physical device discovery is presented in Algorithm 1. The components responsible for extracting the basic information from a device are the *discovery plugins*.
3. Based on the retrieved device model, the discovery framework first checks whether it can find the respective model description in the local repository of known models. If it fails, and assuming that an Internet access is available, it can go on by looking in public, external repositories of device models. If the appropriate device model cannot be found either in the local or remote repository, then automatic integration is not possible, and manual intervention is required in order to connect the device. The new device model description, if available, should be added to the semantic repository so that instances of the same model can be automatically treated in the future.
4. Having retrieved the appropriate device description, the discovery framework now holds all necessary information about the specific device, i.e., the specification of the services and actions offered by the device specified in UPnP. Thus, it can generate a unique UPnP instance-level representation of the device, by using the device's serial number, and further wrap it into the OSGi framework as a bundle. Authorized clients receive a notification about the appearance of the new device instance, and can control it in accordance with its specification.

5. The OSGi bundle is installed and started and, as part of the process, UPnP services are created. The OSGi bundle announces itself in the local network and the *pervasive controller* discovers it using the SSDP UPnP protocol (UPnP Discovery). The devices advertise their integration in the network sending multicast messages when they are added or refreshed.

The *pervasive controller* also sends periodical multicast messages searching for new devices in the network.

1. Once a device is discovered, the device description including its services is exchanged between the UPnP device and the *pervasive controller*.
2. Once the device is fully discovered at UPnP level, the *pervasive controller* sends the information to the *semantic repository*, where the upper layers of the middleware will query (semantically) for services and devices to fulfill the requirements of the goals and plans to be carried out.
3. The *pervasive controller* takes care of the UPnP eventing process, based on the GENA protocol. The *pervasive controller* subscribes to the changes in state variables in the devices and forwards them to the eventing hub in a standard way. The previous steps are depicted in Figure 4.

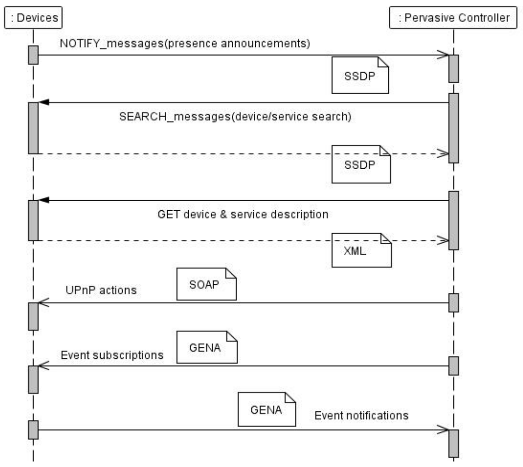
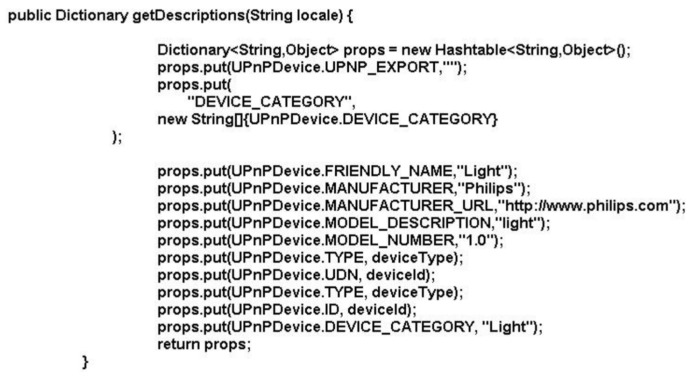

: Devices
: Pervasive Controller
NOTIFY_messages(presence announcements)
SSDP
SEARCH_messages(device/service search)
SSDP
GET device & service description
XML
UPnP actions
SOAP
Event subscriptions
GENA
GENA
Event notifications

Figure 4: Pervasive Controller Sequence Diagram

The semantic discovery control flow is presented in the Algorithm 2. The home designer puts in the OSGi framework the most standard and highly used device types expects to be useful for the user. The home designer also puts the drivers for the most well-known communication technologies, if the user want to manually install a device instance of type not in OSGi manually. However, all subsequent instances of the same device types are automatically installed. After having a successful semantic discovery, the system is able to load and configure a software component called OSGi bundle, and translating the service invocations to the device's specific communication technology.

The *device abstraction layer* in Figure 2 abstracts away the underlying device technology layer above it. For each communication technology (Bluetooth, UPnP, ZigBee) that the system supports, a driver must be available. The responsibility of the driver is to wrap



```
public Dictionary getDescriptions(String locale) {

                Dictionary<String,Object> props = new Hashtable<String,Object>();
                props.put(UPnPDevice.UPNP_EXPORT,"");
                props.put(
                    "DEVICE_CATEGORY",
                new String[]{UPnPDevice.DEVICE_CATEGORY}
        );

                props.put(UPnPDevice.FRIENDLY_NAME,"Light");
                props.put(UPnPDevice.MANUFACTURER,"Philips");
                props.put(UPnPDevice.MANUFACTURER_URL,"http://www.philips.com");
                props.put(UPnPDevice.MODEL_DESCRIPTION,"light");
                props.put(UPnPDevice.MODEL_NUMBER,"1.0");
                props.put(UPnPDevice.TYPE, deviceType);
                props.put(UPnPDevice.UDN, deviceId);
                props.put(UPnPDevice.TYPE, deviceType);
                props.put(UPnPDevice.ID, deviceId);
                props.put(UPnPDevice.DEVICE_CATEGORY, "Light");
                return props;
    }
```

Figure 5: An example of a device model

devices as instances of the *UPnPDevice*[3] interface and register them as OSGi services. This allows the home controller to communicate transparently with heterogeneous devices.

ALGORITHM 1: Physical device discovery

```
 1: Preconditions: The pervasive layer is active and scanning for devices
 2: if new device appears in the network then
 3:     Extract basic information such as device type
 4:     if new device = Bluetooth || ZigBee || UPnP then
 5:         Extract all information the device is capable of providing i.e., device id, address, model
 6:         if device model type description exists in the Semantic Repository then
 7:             add device -> do Sematic device discovery
 8:         else
 9:             device can not be resolved with provided information
10:         end if
11:     else
12:         Device type is unknown
13:     end if
14: else
15:     start scanning for new devices
16: end if
```

The *discovery framework* has been designed to be extensible, new technology modules could be introduced in the system dynamically. This would easily allow developers to introduce new communication technologies in the *pervasive layer* that would appear in the future or integrate an already existing technology for devices.

5.2 Discovery Plugins

The Discovery framework is composed by discovery plugins, one per each communication technology integrated in the middleware framework for discovery of heterogeneous devices. The plugins scan the network in order to search for new devices. The device discovery plugins are responsible for translating low-level or hardware states and activities of the devices (a window, a light, etc.) into events and obtain further information about the newly discovered device. The respective device plugin for the discovery framework reads

[3]org.osgi.service.upnp.UPnPDevice, http://www.osgi.org/javadoc/r4v42 /org/osgi/service/upnp/UPnPDevice.html

ALGORITHM 2: Semantic device discovery

```
 1: Preconditions: A device has been physically discovered and its device model has been retrieved
    from the Semantic repository
 2: if proxy exists in the semantic repository then
 3:     Proxy instance is downloaded into the middleware
 4:     New proxy object is created
 5:     Proxy object is configured for particular discovered device
 6:     Pervasive controller exposes the device services as UPnP
 7:     Device is exposed as SM4All device
 8: else
 9:     Proxy could not be downloaded, installed or created
10: end if
```

all available information from the device. Which information the device provides mainly depends on the technology used, e.g., Bluetooth. Typically, information about the device name, type and possibly, about the services provided by the device can be obtained. The information is used to perform the reasoning towards the OSGi registry to infer the type of device and the services it provides, and it returns a device model, containing the device id, type, name, services, etc. An example of a device model is shown in Figure 5. Each plugin must extract as much information as it can about the discovered device. The richness of this information depends on the communication standard.

Figure 6 shows the required sequence of actions for the device discovery process to add a new heterogeneous device. The pervasive layer is active and scanning for new devices appears in the network. When a new device announce itself in the network, discovery framework confirms the discovery plugin for the new device communication protocol. The discovery framework extracts the basic information from the device, otherwise needs to add a new discovery plugin for new communication protocol. The discovery plugin sends this information to the semantic repository, then semantic repository finds a suitable device type description based on the extracted information and returns this information to the discovery plugin. If not a suitable device type description can be found, the device is marked as unknown to the system.

In Section 8, Bluetooth and ZigBee plugins are explain as these technologies are suitable for low data rate applications with limited battery power. Each plugin has its own properties and is properly designed according to the correspondent communication protocol. The *home gateway* is loaded with Bluetooth and ZigBee adapters. This device is called base node in the context of device discovery. It always has a Bluetooth and ZigBee client applications for incoming connections within the home network. These client applications are part of discovery framework and always active and scanning for new devices periodically and establishes communication with the middleware layer.

6 Pervasive Controller

The main goal of the *pervasive controller* is to manage the devices that run in the local network, making them known to the upper layers of the middleware framework, so they can access the services provided by the devices. The access to the services is done using UPnP interfaces (based on the SOAP protocol for the message interchange) as shown in the Figure 3. The *pervasive controller* is an extended UPnP Control Point for UPnP devices

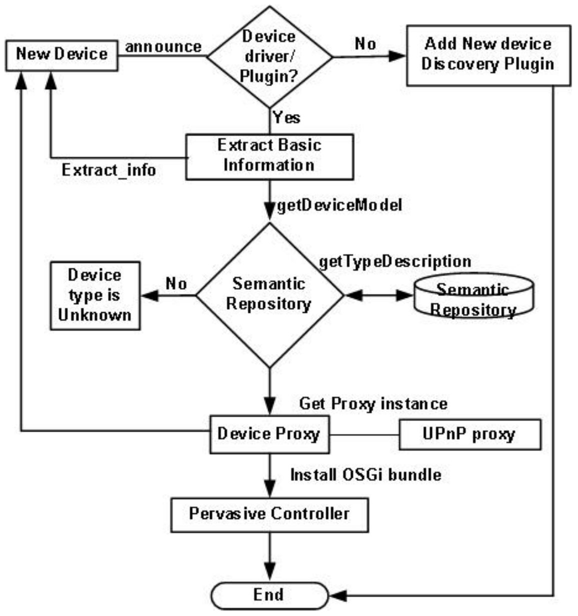


Figure 6: A New device addition

(native or proxies) that provides a common access for the upper middleware layers and applications to the UPnP services provided, feeding the system with updated information about the existing devices and services in the system.

Once the device has been physically discovered and semantically resolved, and the OSGi bundle with its UPnP interfaces is created, the *pervasive controller* is able to perform the UPnP discovery in the network, maintaining an internal and up to date registry of services and devices in the local network. The *pervasive controller* is responsible for:

- Keeping the descriptions of the services provided by the OSGi bundles that control the devices in the network.
- Informing the OSGi framework about the addition and removal of services and devices in the network.
- Taking care of event managing and sending the events to the *composition layer* or other subscribed clients in a standard way.

The layer pattern it is using ensures device and platform independence, which primarily means that device and platform precise details need to be abstracted away. Another important feature is eventing, and we overcome this challenge using publish/subscribe based patterns [4].

6.1 Contract-First Service Development

Herein, we follow the contract-first development paradigm for the Web Services development, i.e. we first start with the WSDL contract, and use Java to implement the prescribed contract. This way, one can guarantee among others contract ability for long time, independence from SOAP stacks particularities, re-usability, and good response times. This is useful for the interactions between the home gateway which exposes the WS functionalities and the middleware client. To support eventing there need to be two-way communication between the pervasive controller and the middleware clients. This means, that clients must be able to interact with the pervasive controller server, written in Java, independently of their technology (e.g. the visualization tool is written in Ruby), but also the other way around.

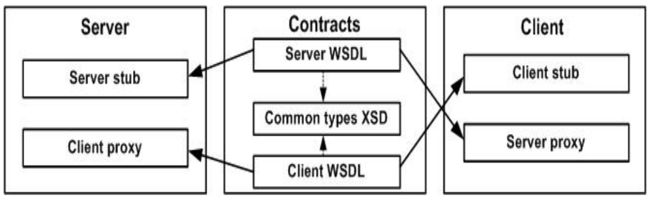


Figure 7: Contract and code generation process

The code generation scheme for the two-way communication using contract first development is shown in Figure 7. There are two services described by WSDL files which share their data types via a XSD file which they both import. For the server side (pervasive layer), a stub is generated from the server WSDL and proxy is generated from the client WSDL. The stub of the server WSDL is used to implement operations to register, control and subscribe to services. A client proxy is instantiated for each registered client and used for further communication with that client. For the client side (middleware clients) a stub is generated of the client WSDL and a proxy for the server WSDL. The stub is used to implement the logic for handling events from the pervasive layer. The proxy is used to register, control and subscribe to services in the pervasive layer.

6.2 Publish/Subscribe

The pervasive controller uses the publish/subscribe pattern to decouple the UPnP services (the producers) and the client applications (the consumers). This decoupling can take place in time, space and synchronization [10]. There are many publish/subscribe based patterns, all offering specific advantages. We used two variations in the architecture of the pervasive layer, such as listener and eventing.

The listener pattern is a simple pattern for publish/subscribe based eventing. The event-driven communication between the pervasive controller and the middleware clients is also based on the listener pattern. In the middleware framework, the whiteboard [2] pattern is used for the event-based communication between the UPnP importer/exporter and the controller.

Eventing is an important feature in the proposed framework. The proposed architecture supports event driven communication as it is more efficient and scalable than communication based on polling [22]. The eventing support of pervasive layer is based on the UPnP eventing mechanism. This is because all heterogeneous devices are abstracted to UPnP in the device abstraction layer. UPnP eventing is based on state variables, which represent a visible state of a UPnP service. Using a publish/subscribe based technology, interested parties can register to such state-variables. If such a state-variable changes, then registered parties receive a notification.

Figure 8 shows the flow of an event which starts as an UPnP state-variable that is changed, passes through the layers and eventually reaches the web service clients. As soon as the state-variable changes to which a client is subscribed (step 0), the following steps take place:

1. A UPnP device sends an event using the General Event Notification Architecture (GENA) to subscribers, including the UPnP service wrapper component of the pervasive layer.
2. The UPnP service wrapper looks up event listeners in the OSGi service registry to initiate OSGi eventing using the whiteboard pattern [2].
3. The UPnP service wrapper propagates the event to the *controller*.
4. The *controller* running in the OSGi framework notifies the *server* that there is a separate process outside the OSGi framework.
5. The *server* invokes the stateVariableChanged operation (defined in the clients WSDL) for registered clients.

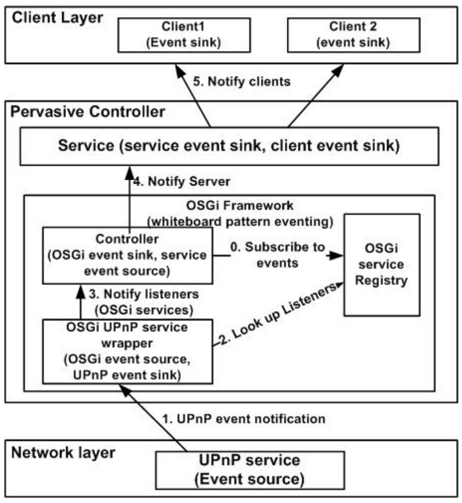

Client Layer
Client1 (Event sink)
Client 2 (event sink)
5. Notify clients
Pervasive Controller
Service (service event sink, client event sink)
4. Notify Server
OSGi Framework (whiteboard pattern eventing)
Controller (OSGi event sink, service event source)
0. Subscribe to events
OSGi service Registry
3. Notify listeners (OSGi services)
2. Look up Listeners
OSGi UPnP service wrapper (OSGi event source, UPnP event sink)
1. UPnP event notification
Network layer
UPnP service (Event source)

Figure 8: Event propagation

6.3 Implementation

The are three main components of the pervasive controller that can be distinguished: the *WS gateway, the home controller* and *virtual devices*. The home controller and virtual devices are OSGi bundles. The WS gateway is a web service that middleware clients can use to interact with the pervasive controller. The home controller is responsible for the communication within the OSGi framework to the WS gateway. The proposed middleware platform runs on the *Apache Felix*[4] implementation of the OSGi framework. Its important features include eventing and the device and platform independence.

6.3.1 WS Gateway

The *WS gateway* is a web service which allows clients to control services and receive notifications of events. It advertises the following operations to clients through a WSDL interface: (de)registerClient, (un)subscribe, listServices and invokeAction. It is written in Java and uses the *Apache CXF*[5] framework to run the web service and to generate Java

[4]http://felix.apache.org
[5]http://cxf.apache.org/

code from the WSDL files. The *WS gateway* is the glue between middleware clients and the pervasive controller. It administers the communication with the controller, keeps track of registered middleware clients and propagates events from the home controller to the registered clients.

A web service can support any of the following three styles: a Remote Procedure Call (RPC), a document-oriented style that is associated with Representational State Transfer (REST), or a hybrid style that is called REST-RPC. We considered two styles that are used mostly on the modern web to communicate with the outside world, such as Rest and the RPC style. The communication with the home controller happens via message passing over a socket.

6.3.2 Home Controller

The home controller is an OSGi bundle that is responsible for handling eventing and controlling the UPnP services available in the OSGi framework. It functions as a bridge between the OSGi layer and the web *WS gateway.* It uses message passing over a socket to communicate with the pervasive server. The advantage is that it is not necessary to port Apache CXF framework with all its dependencies to an OSGi bundle. With regard to eventing, the home controller acts as a bridge between the eventing based on the whiteboard pattern [2] that is used in the OSGi framework, and the eventing based on the listener pattern that takes places in the *WS gateway.*

6.3.3 Virtual devices

Virtual devices are devices that are implemented only as software and have no real hardware. Virtual devices can be useful for testing and demonstration purposes. They can be implemented by using the UPnP interfaces defined by the OSGi framework. After implementing these interfaces, the virtual devices can be deployed in the OSGi framework as bundles. If these bundles are activated, the virtual devices function as real devices in the system.

7 Middleware Clients

The middleware client layer provides the means to the final users to interact with the middleware and control several devices. The clients may be a visualization and simulation tool, a user interface, or an intelligent composition agent. We introduce a middleware client layer above the middleware to interact with rest of the framework as shown in Figure 2. The potential clients communicate with the pervasive controller at the middleware layer through the exchange of SOAP messages as shown in the Figure 10. Currently, there are three extensions implemented for the pervasive layer as a middleware client: a home visualization and simulation [16] (RuG ViSi Tool), a service composition planner [14] and user interfaces [7]. All are implemented as middleware clients, but for different purposes.

7.1 RuG ViSi Tool

Testing and verifying the behaviour of such large pervasive systems is costly, setup time in a real home is high, and tuning the physical devices consumes a great deal of time. Therefore, an environment that mimics as closely as possible the real setting and is able to simulate a number of interactions and behaviours can greatly help the development and testing of the proposed middleware framework with different communication technologies. It also serves the purpose of acceptability testing with potential users of the system. More details about the architecture and implementation of the RUG ViSi tool can be found in Appendix A

7.2 Service Composition Planner

To satisfy the wishes of the home users and guarantee their comfort and safety, the house has to be able to exhibit quite complex functionalities rather than just being able of triggering some single service. To this end the ability to construct and execute compositions of individual services is of paramount importance. To this end, one can resort to standard Business Process descriptions and execution engines such as BPEL [16] or employ more intelligent approaches that support automatic, dynamic and customizable compositions. In [14] such a flexible composition component that bases on Artificial Intelligence domain-independent planning has been proposed, to generate context-aware compositions at runtime, in a constantly evolving environment. The abstraction and unified view-description of all available devices offered by the pervasive layer as already described allows the composition planning component to automatically build the necessary compositions without caring about the heterogeneity in implementation details or data.

The composition engibe bases upon a domain-independent planner, which ultimately models the home domain as a Constraint Satisfaction Problem (CSP). The planner is equipped with a number of special features that go beyond classical planning and are of particular importance to the requirements associated with modeling and controlling service interactions in a smart home environment. Firstly, it supports efficient handling of variables ranging over large domains, which are commonly used by smart home components (temperature measurements, TV channels, the locations monitored by the location component are all essential pieces of information). Secondly, the representation of the home domain as a dynamic constraint network, which allows connecting and disconnecting constraints on-the-fly, enables the efficient update of the current environmental state as delivered by the context-awareness model, without the need of re-loading the whole domain. Another important characteristic of the planner, which makes it particularly wellsuited for adaptable and user-centric environments, is that it accommodates for a high level language for expressing extended goals [13]. The nature of the goal language is declarative, decoupled from the procedural and operational details of the services. The service operations that fulfill the properties prescibed in the goal are synthesized by the planner automatically and at runtime, depending on the current state of the environment and the devices available at the moment. More details about the planner and the techniques it uses can be found in [13].

The planner takes the following ingredients as input:

- The representation of the home in the form of a planning domain, i.e. the description of the available service operations as *actions*, in terms of preconditions and effects. This representation of service operations as actions is stored in the Semantic Repository, as part of the information about each available service.
- The description of the current state of the home as delivered by the Context Awareness component. The planner subscribes to all state variables it is interested in, and is thus asynchronously notified whenever their value changes. Whenever it receives a change

event, it changes its initial state accordingly.

- A goal prescribing a set of properties to be achieved. Goals are stored in the Semantic Repository, and can be requested either by the User Interface or by the Rule engine.

Given the goal and the description of the domain instance, the planner computes a plan, i.e. a partially ordered set of actions which have the potential to satisfy the goal. The planner executes the plan step-by-step, by asking from the Invocation Dispatcher to execute the respective waiting to be informed about its outcome. If the response indicates a failure, then the planner will perform re-planning, i.e. it will come up with an updated plan starting from the new initial state as delivered by the Context Awareness component. If the failure indicates that a service is permanently out of order, then the plan will look whether the same goal can be satisfied by an alternative plan, without using the defective appliance.

In the followings we provide an example of how the semantic description of the three operations (one for each of the possible positions of the bed) offered by the service “MotorizedBed”, represented in xml format, looks like:

```
<service name="MotorizedBed">
 <variable name="bed_position" type="BedPosition"/>
 <operation name="setHigh">
  <pre>
      <eq-val var="bedPosition" value="MIDDLE"/>
  </pre>
  <assign-val var="bedPosition" value="HIGH"/>
 </operation>
 <operation name="setLow">
  <pre>
      <eq-val var="bedPosition" value="MIDDLE"/>
  </pre>
  <assign-val var="bedPosition" value="LOW"/>
 </operation>
 <operation name="setMiddle">
  <pre>
      <neq-val var="bedPosition" value="MIDDLE"/>
  </pre>
  <assign-val var="bedPosition" value="MIDDLE"/>
 </operation>
</service>
```

A set of predefined goals depending on the user’s routine and needs are stored in the Semantic Repository and made available in the supported UIs. In the followings we present how the “good night” goal described in the example described Section 1 looks like. The `achieve-maint` construct prescribes that the propositions which follow should be satisfied in some future state and remain true till the end of the plan (more about the goal language van be found in [14]).

```
<achieve-maint>
  <eq-val var="MotorizedBed_bedroom1Bed_bedPosition" value="LOW" />
 <eq-val var="Lights_bedroom1Light01_lightStatus" value="OFF" />
  <eq-val var="Lights_bedroom1Light02_lightStatus" value="OFF" />
 <eq-val var="ALARM_bedroom1Alarm_alarmClockTime" value="08:00"/>
</achieve-maint>
```

.

7.3 User Interface

The user interface provides the means for the finals users to interact with and control the devices. The basic module of the user layer is the Abstract Adaptive Interface (AAI) [7], which acts as a proxy that provides services to the particular User Interfaces (UIs) that can be connected to the system. Its task is to collect information about the available service operations of active devices and the goals kept in the repository, and forwards them to the concrete UIs, with which it interacts via the exchange of SOAP messages. The information collected from the repository includes visual data (icons) associated with the devices, as well as information about the location of the devices, so that they can be organized accordingly, depending on the capabilities of the concrete UIs. Moreover, the AAI is seamlessly updated to reflect the most recent status of the devices and notifies the concrete UIs connected to the system accordingly. Its implementation is based on *Apache Tomcat*[6] and *Apache Axis*[7].

8 Implementation, Experiments and Validation

To evaluate whether the middleware framework is effective and the used techniques have adequate performance, a technical evaluation of the system is described next. The significance of our proposed work lies mainly in the heterogeneous device discovery and middleware framework based on OSGi and UPnP for smart homes. The former provides highly dynamic, extendable and open discovery framework to achieve heterogeneity, in which devices join and leave while adopting different communication technologies, while the later achieves interoperability among devices and multi-platforms. We first describe the application scenario to address those two significant features, and then experimental setup of the different possible tests to evaluate the system performance as well as the results of the experiments.

8.1 Case Study Evaluation

We assure that the *pervasive layer* can be used to integrate heterogeneous devices in a middleware framework and that the upper layers can use the device's services without having information about devices low level communication technologies. Let's have a look at the necessary steps required in the proposed architecture to fully integrate a device which appear in the network layer. Section 3.1 explains a scenario to add a TV in smart homes that has Wireless Home Digital Interface (WHDI), we take it as an example and explain the operations required at the middleware framework. Figure 6 shows the required sequence of actions for the device discovery process to add a new heterogeneous device.

8.1.1 Pervasive Layer Operation in the Case Study

The *pervasive layer* is active and scanning for devices on the *home gateway*. When the TV turns ON and appears in the network layer, the WHDI adapter, part of the *discovery framework*, extracts some basic information such as device id, type and unique address from the device's profile by using extractBasicInfo and getDeviceModel method. There are

[6]http://tomcat.apache.org/download-70.cgi
[7]http://axis.apache.org/axis2/java/core/

several possibilities when device cannot be discovered physically, such as: the device cannot be discovered because its communication technology is not supported by the *pervasive layer*, the device type is unknown, the device cannot be resolved with the information provided and the device model of the discovered device could not be retrieved.

The *home gateway* is loaded with a WHDI adapter and discovery plugin to discover TV. To integrate a new device and to achieve interoperability, well-known protocol discovery plugins and adapters must be available at *home gateway.* As the number of protocols grow, further development effort is required. However, our experiences of developing the case study indicate that this effort is not infeasible; each protocol needs to be developed only once. Generally, each new protocol required almost one month development time for a single experienced Java developer.

Once the device information is known, the WHDI discovery plugin queries the semantic repository to decide on the device type using getTypeDesrciption method. The result of the query is a link to a proxy (device type description) residing in the *semantic repository*, and a proper OSGi bundle is downloaded and installed at the pervasive controller that can handle the communication with the device and offer the UPnP interfaces. This OSGi bundle will be capable to control the TV. An instance of this particular device is added as an OSGi bundle in the OSGi framework. The OSGi bundle is paired with the device and discovered by the *pervasive controller* (UPnP Control Point) as a UPnP device. This way the services offered by the TV are register in the OSGi registry, and become available for the upper middleware layers. The middleware clients at upper layers can access the services offered by the UPnP proxy as UPnP clients. The *pervasive layer* has a real time knowledge about the status of the device and can detect any change in the status of the device or its services and notify this change to the rest of the middleware layers.

8.2 System Setup

The *middleware framework* architecture is fully implemented to test its technical properties. A prototype of the *home gateway* system was built on a laptop equipped with a *Bluetooth* and *ZigBee* using an open source OSGi framework and UPnP protocol with Java (Figure 9). The core software modules of the proposed architecture, including, the discovery plugins (Bluetooth and ZigBee), device abstraction layer, discovery framework, semantic repository, pervasive controller (WS Gateway, home controller, virtual devices) and middleware clients (RUG ViSi tool, service composition planner and user interfaces) were implemented.

8.2.1 Bluetooth Device Discovery Plugin

The *home gateway* is loaded with a Bluetooth adapter (BT discovery). This device is called base node in the context of Bluetooth discovery. The *home gateway* always has a Bluetooth client application for incoming connections within the home network. Bluetooth discovery is a part of the PDF. It is always active and scanning for devices, and establishes communication with the middleware layer.

The proposed architecture is implemented for two kind of devices: a desktop system with Java Standard Edition and a Java Mobile Edition (1.2 J2ME) running on a mobile phone (Nokia N85 - Symbian OS 9.3, Bluetooth v2.0 with A2DP). We have split the *home gateway* software part into a Bluetooth generic interface library and a Bluetooth device discovery specific part (BluetoothDiscovery). A simulated UPnP lamp device is registered

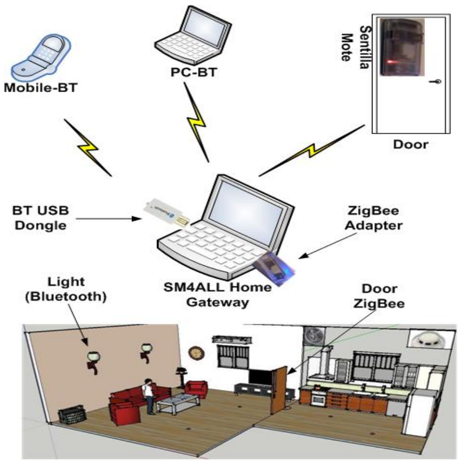


Figure 9: System Configuration

to the UPnP network as an OSGi bundle. A Bluetooth device that has been discovered is registered in the middleware. When the device is properly configured, it is installed at the OSGi framework. Thus, a lamp UPnP representation can control the discovered physical Bluetooth device.

The experimental setup is based on using a 2.66 Ghz computer running Windows Vista 32 bits and Java 1.6.0 for Bluetooth desktop application and a Nokia N85 mobile phone running Java Mobile Edition. Another computer using a 2.66 Ghz running Windows Vista 32 bits and Java 1.6.0 as a *home gateway.* One Sentilla JCreate USB adapter is attached with *home gateway.* Two USB Bluetooth (Bluetooth version 2.0 with Enhanced Data Rate) sticks, one for *home gateway* (PC A) and another for a desktop application which is a Bluetooth remote control (PC B) to control devices connected with *home gateway.* We used Java APIs for Bluetooth (JSR 82) on the N85 and *apache felix* UPnP or any UPnP network scanner. The phone is exported as an OSGi bundle.

After the bootstrap, the Bluetooth discovery framework is turns on and also the Bluetooth client application (Java application) on *PC A*. It start *apache felix* or any UPnP network scanner that is searches for devices to nearby *home gateway* within smart environments. A new device will be discovered, and at this point device information is extracted, e.g., friendly name (BluetoothLamp) and also a query is made to UPnP controller to configure the device OSGi bundle. Device OSGi bundle (BluetoothLampUPnPDevice) is installed at UPnP controller. After having a successful Bluetooth connection between *PC B* and *home gateway*, then Bluetooth adapter on *PC B* can be used as a remote control to setting *ON* and *OFF* some other device such as lamp in RuG ViSi tool connected to PC A (home gateway).

8.2.2 ZigBee Device discovery Plugin

ZigBee is a global standard for wireless communication and particularly adopted for use in home control automation. A ZigBee device plugin is developed to dynamically integrate ZigBee devices in the pervasive controller. The ZigBee supported device that is selected for the experimental work is *Sentilla mote* [23], as shown in the Figure 9. It is chosen on the basis of their innovative potential and the large number of new applications they can support. A single *Sentilla mote* contains the following elements: 3-axis accelerometer, a temperature sensor, eight LEDs, two standard connectors (analog inputs), and an expansion connector. Communication between mobile devices and host PC (*home gateway*) is realized using the *USB Gateway* device. A laptop is equipped with a *Sentilla* network adapter and running the network-monitor inside the Sentilla environment.

The Sentilla mote accelerometer provides us three values representing the acceleration in the x-, y- and z-axis. By implementing the *AccelSensorMonitor* method in the Read-Sensor class of client application, the sensor data is captured and processed. A simulated UPnP door device is registered to the UPnP network as an OSGi bundle to test the ZigBee discovery module as shown in Figure 9. A *ZigBee Sentilla* mote that is coupled with the discovered door is registered in the middleware. When the device is properly configured, it is installed on the OSGi framework. Thus, a door UPnP representation can control the discovered physical ZigBee device. Whenever the door is open, the accelerometer sensor y-axis and UPnP device values change which can be visualized in simulation. We evaluate its performance on a real testbed of *Sentilla JCreate* wireless nodes.

After having successful installation of heterogeneous devices, such as Bluetooth and Zig-Bee at home network. The corresponding discovery plugin bundle automatically import the devices and services from each different network device into OSGi framework. All services offered by the discovered devices are available at the OSGi service registry.

8.3 Bluetooth proxy tests

The visualization tool can be used for initial tests, and provides a first indication of the system's behaviour and performance in real world scenarios. Reaction to events is nearly instantaneous for both real and simulated devices.

The purpose of these tests is to analyze the behaviour of the pervasive layer with Blue-tooth devices, in particular in terms of *good-put, throughput, round trip time* and *packet error rate*. To investigate the performance of a Bluetooth device proxy, we conducted a series of tests are conducted using the measurement setup explained above. *Good-put* is the application level throughput, i.e., the number of useful information bits, delivered by the network to a certain destination, per unit of time. We define *throughput* to be the ratio of the number of error free transmissions between the PC B , where the *home gateway* is running, and PC A which is connected to a Bluetooth adopter in a given time interval. *Packet error rate*, where a single reported packet error could result from one or more bit errors occurring in the packet, the bit error rate provides additional information, reporting the total number of errors occurring in the demodulated packets. The term *round-trip time* represents the elapsed time for the transit of a signal over a closed-circuit. A web service call is made from PC B and is sent to the receiver PC A, and PC A again sends back the response spontaneously to the PC B. The connection is established across the L2CAP (Logical Link Control and Adaptation Protocol) layer [27]. On the basis of this layer, we add time stamps to each web service request/response invocation at both PC A and PC B.

The measurements are taken five times for the physical device running on PC A. The total time interval is set to one minute. The results are shown in the Table 1.

Round trip time varies between 35 and 260 ms. The packet loss and packet error rate results are not predictable normally in our current experimental setup because testing is done at the application layer instead of lower layers, such as RFCOMM. In most cases, the packet error rate increases when the distance of the communicating devices increases. We observe that the good-put is directly proportional to the offered load. The good-put varies

Calls one min.	Throughput kbps	Avg. Round-Trip time ms	Avg. Goodput kbps
22	723	76	550
21	723	105	579
19	723	98	560
20	723	80	598
26	723	92	584
Average	723	90,2	574,2

Table 1: Bluetooth proxy Test results

between 550 and 600 Kbit/s. Bluetooth supports both asynchronous and synchronous communication. We used asynchronous communication to transfer data between PC A (home gateway) and PC B (BT remote control). It can be seen from Table 1 that we have maximum throughput in all experiments.

9 Conclusions

To satisfy the needs for interoperability and dynamicity, while keeping human intervention to the minimum, we have designed and implemented a layered architecture which abstracts away from the particularities of the lower device levels to a higher level where all devices are exposed in a unified way as services, and an application level on top that allows building more complex applications that utilize and combine different services. More specifically, we have built a middleware that allows the automatic discovery and connection of devices with heterogeneous network protocols, e.g. Bluetooth, ZigBee, etc. by the use of established standards, namely UPnP and OSGi. All that is needed at this level of automatic discovery is deploying a driver once per communication technology. In order to support a high degree of automation and dynamicity at the application level all available devices are exposed as services. To enable the development of more complex functionalities that are based on service compositions, the availability of machine-parsable and standardized descriptions of their provided functionalities becomes paramount. To this end, each vendor should provide a a description of the functionalities offered by each device type in a standardized format. A semantic repository is used to collect and store these descriptions. This way, every time a new device of a supported type is discovered in the home network, all steps of its integration up to the highest levels of abstractions can take place in a purely automated fashion.

We have described a case-study and shown how the components of architecture co-operate to accommodate for the requirements of discovery and integration. We have described an experimental setup to show how the discovery plugins work for two particular protocols, Bluetooth and ZigBee. We have conducted some performance tests regarding the behavior of the Bluetooth device proxy, which verify that the overhead introduced by the adopter is almost negligible.

Although the proposed service-oriented architecture promises a better solution for smart home network, a number issues still remain open, and require further research. One of the most challenging issues that need to be addressed are related to concurrency. For example, what happens if the DVD theater requests the curtain to close, but the illuminometer asks it to open at the same time? At the level of more complex applications, like the parallel execution of service compositions, the number of conflicts increases, and their resolution require special attention. Another issue is adaptation and recovery from failures, that may stem from different levels of the architecture (network failures, device failures, application failures) and are thus due to different causes. For example, device failures in the OSGi environment are due to the inherent properties of ad-hoc and pervasive computing. Application failures on the other hand may occur due to unverified code, mismatched libraries, and even intentionally malicious codes [1]. Network failures occur either at the network or node level because the collected data is faulty due to internal and external influences, such as battery drain, environmental interference, sensor aging [26]. Thus, depending on the different nature of failures that may occur, different, tailored strategies have to be employed to effectively deal with them.

Acknowledgement

Here we thank people from SM4ALL consortium, programmers in Groningen and people who helped/participated in evaluation. The research is supported by the EU project Smart Homes for All, contract FP7-224332.

References

[1] Ahn, H., Oh, H., Sung, C.O.: Towards reliable osgi framework and applications. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 1456–1461. SAC '06, ACM, New York, NY, USA (2006)

[2] Alliance, O.: Osgi alliance, listeners considered harmful, the whiteboard pattern (2004), http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf

[3] Babiloni, F., Cincotti, F., Marciani, M., Salinari, S., Astolfi, L., Aloise, F., Fallani, F.D.V., Mattia, D.: On the use of brainâĂŞcomputer interfaces outside scientific laboratories: Toward an application in domotic environments. In: International Review of Neurobiology, International Review of Neurobiology, vol. 86, pp. 133 – 146. Academic Press (2009)

[4] Baldoni, R., Querzoni, L., Virgillito, A.: Distributed event routing in publish/subscribe communication systems: a survey. Tech. rep., DIS, Universit'a di Roma La Sapienza (2005)

[5] Brumitt, B., Meyers, B., Krumm, J., Kern, A., Shafer, S.A.: Easyliving: Technologies for intelligent environments. In: Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing. pp. 12–29. HUC '00, Springer-Verlag (2000)

[6] Bushmitch, D., Lin, W., Bieszczad, A., Kaplan, A., Papageorgiou, V., Pakstas, A.: A sip-based device communication service for osgi framework. In: First IEEE Consumer Communications and Networking Conference (CCNC). pp. 453–458 (2004)

[7] Catarci, T., Ciccio, C.D., Forte, V., Iacomussi, E., Mecella, M., Santucci, G., Tino, G.: Service composition and advanced user interfaces in the home of tomorrow: The sm4all approach. In: Ambient Media and Systems (AMBI-SYS). pp. 12–19 (2011)

[8] Chang, G., Zhu, C., Ma, M., Zhu, W., Zhu, J.: Implementing a sip-based device communication middleware for osgi framework with extension to wireless networks.

In: First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS). vol. 2, pp. 603 –310 (june 2006)

[9] Dobrev, P., Famolari, D., Kurzke, C., Miller, B.: Device and service discovery in home networks with osgi. Communications Magazine, IEEE 40(8), 86 – 92 (aug 2002)

[10] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114 – 131 (2003)

[11] Gouvas, P., Bouras, T., Mentzas, G.: An osgi-based semantic service-oriented device architecture (2007)

[12] Helal, S., Mann, W., El-Zabadani, H., King, J., Kaddoura, Y., Jansen, E.: The gator tech smart house: A programmable pervasive space. Computer 38, 50–60 (2005)

[13] Kaldeli, E., Lazovik, A., Aiello, M.: Extended goals for composing services. In: Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009). AAAI Press (2009)

[14] Kaldeli, E., Warriach, E.U., Bresser, J., Lazovik, A., Aiello, M.: Interoperation, composition and simulation of services at home. In: 8th Int. Conf. on Service Oriented Computing (ICSOC-10). vol. LNCS 6470, pp. 167–181. Springer (2010)

[15] Kim, D.S., Lee, J.M., Kwon, W.H., Yuh, I.K.: Design and implementation of home network systems using upnp middleware for networked appliances. Consumer Electronics, IEEE Transactions on 48(4), 963 – 972 (nov 2002)

[16] Lazovik, E., den Dulk, A., de Groote, M., Lazovik, A., Aiello, M.: Services inside the Smart Home: A Simulation and Visualization tool. In: 7th Int. Conf. on Service Oriented Computing (ICSOC-09). pp. 651 –652 (2009), `http://www.cs.rug.nl/~aiellom/publications/icsoc09.pdf`

[17] Li, X., Zhang, W.: The design and implementation of home network system using osgi compliant middleware. Consumer Electronics, IEEE Transactions on 50(2), 528 – 534 (may 2004)

[18] Marples, D.: The open services gateway initiative: An introductory overview (2002)

[19] Mozer, M., Dodier, R., Miller, D., Anderson, M., Anderson, J., Bertini, D., Bronder, M., Colagrosso, M., , Cruickshank, R.: The adaptive house. In: IEEE Seminar Digests (2005)

[20] Ngo, L.: Service-oriented architecture for home networks. In: Seminar on Internet working. pp. 1 – 6 (2007)

[21] Service platform core specification v.4 (2010), www.osgi.org

[22] Pietzuch, P., Bacon, J.: Hermes: a distributed event-based middleware architecture. In: Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on. pp. 611 – 618 (2002)

[23] (2010), http://www.sentilla.com

[24] Song, H., Kim, D., Lee, K., Sung, J.: UPnP-Based Sensor Network Management Architecture. In: Second International Conference on Mobile Computing and Ubiquitous Networking (ICMU) (2005)

[25] Upnp device architecture version 1.1. (2008), http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf

[26] Warriach, E., Tei, K., Tuan, A.N., Aiello, M.: Fault detection in wireless sensor net-

works: a hybrid approach. In: 11th ACM Conference on Information Processing in Sensor Networks - POSTER Session (2012)

[27] Warriach, E., Witte, S.: Approach for performance investigation of different bluetooth modules and communication modes. In: Emerging Technologies, 2008. ICET 2008. 4th International Conference on. pp. 167 –171 (oct 2008)

[28] Wu, C.L., Liao, C.F., Fu, L.C.: Service-oriented smart-home architecture based on osgi and mobile-agent technology. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 37(2), 193 –205 (march 2007)

APPENDIX

A RUG ViSi Tool

The challenge is to create a real-life simulation of a home in which home applications are integrated as much as possible. The aim is to show that all heterogeneous networks and devices (sensors and actuators) with different services and communication technologies seamlessly integrate into the middleware, provide devices' services and information using a common and standard abstraction interface, and can communicate with each other. The home instance visualized and controlled by the users is based on a virtual reconstruction of a real apartment built at the premises of the Fondazione Santa Lucia (FSL) in Rome, a hospital specialising in neuromotor rehabilitation (www. hsantalucia.it). The apartment consists of four rooms (two bedrooms, a kitchen and a living room), equipped with 32 simulated devices (lights, doors, motorized bed, curtains, windows, TV, air condition etc). The apartment modeled at the visualization platform (RuG ViSi tool) is equipped with virtual devices implemented in Ruby[8]. Those simulated devices are coupled with the UPnP devices at the pervasive layer. This way, whenever a device state is changed, the result is reflected in real time at the visualization layer, and thus the effects of the instructions issued by the UI can thus be visualized. Figure 10 depicts the interaction between the UI, the pervasive layer and the RuG ViSi Tool visualization platform.

A 3D visualization platform based on Google SketchUp [16] is integrated in the framework, as a middleware client of the pervasive controller shown in Figure 2, to simulate as closely as possible the layout and behavior of a real home. It is upgraded by adding support for eventing. The visual model of the house is updated in a event driven manner using the Ruby.

Appliances, sensors and actuators that we wish to simulate are coupled with simulated UPnP devices. These devices, thus, live outside the RuG ViSi tool and interact with it as if they were actual physical devices. Figure 10 illustrates the virtual house and a number of UPnP devices: a door controller, a light, a window controllers, and a television set. On the bottom, there is also a UPnP module (Actor), which represent the position of a user in the house. This can be coupled with a location detector of the house that gives to the home constantly information over the position of the user. In our simulation, we use the module to virtually move people inside the home.

The RuG ViSi tool can, naturally, be also coupled with real physical hardware devices. For instance, we have coupled a door service with a *sentilla* mote equipped with an ac-

[8]http://www.ruby-lang.org

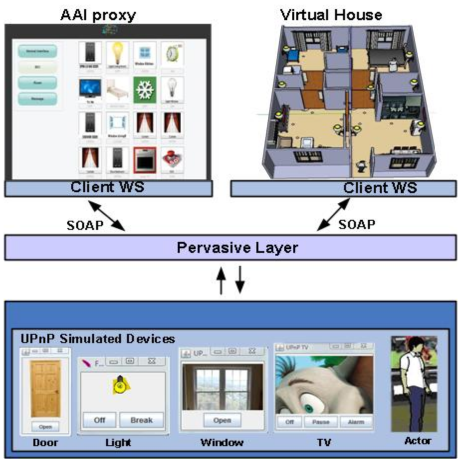


Figure 10: General Architecture of the RuG ViSi tool

celerometer and a radio connection. The hardware is plugged in the OSGi layer and, when shaken, triggers a state change. In other words, it opens and closes a specific door in the virtual home. We have also coupled a light service with a Bluetooth enabled mobile. The hardware is plugged in the OSGi layer and change the state of the light by sending *ON* and *OFF* commands. In other words, it turns *ON* and *OFF* a specific light in the virtual home. The architecture of the RuG ViSi tool is an instance of the general one proposed in Section 3: it mainly covers the pervasive layer, allowing for dynamic join/leave of UPnP compliant and UPnP proxies devices. It provides translation from the low-level invocation into high-level web service invocations and call-backs which can then be orchestrated and visualized in a 3D rendering environment as shown in Figure 10.