A Service-Oriented middleware for device interoperation in the smart home

(DRAFT)

Jaap Bresser

Advisor: Marco Aiello
Co-advisor: Alex Telea

August 23, 2010
Abstract

Pervasive computing environments such as our future homes are the prototypical example of a dynamic, complex system where Service-Oriented Computing techniques will play an important role. A home equipped with heterogeneous devices, whose services and location constantly change, needs to behave as a coherent system supporting its inhabitants. Achieving interoperability between devices in such an environment is a challenge.

In this thesis we propose the use of a Service-Oriented middleware to face this challenge. We present a fully implemented architecture for domotic applications which uses the concept of a service as its fundamental abstraction.

An important component of the architecture of this system is the pervasive layer. The pervasive layer uses a Service-Oriented architecture based on standardized technologies such as OSGi and UPnP to achieve interoperability between devices. We will describe the architecture and implementation of the pervasive layer, and evaluate both its usability and its performance. We evaluate the usability of the system using a visualization of simulation environment and the performance using a stress-test tool. We will also briefly describe the other parts of the architecture to give an indication of the value of the system as a whole.
5.2.1 Contract first service development ........................................ 28
5.2.2 Client server interaction ...................................................... 28
5.2.3 Client implementations ....................................................... 31

6 Evaluation .............................................................................. 34

6.1 Performance test ................................................................... 34
6.1.1 Latency .............................................................................. 35
6.1.2 Test method ........................................................................ 34

6.2 Test results ........................................................................... 34
6.2.1 Analysis .............................................................................. 35

7 Conclusion ............................................................................. 36
Chapter 1

Introduction

The vision of the Internet of Things brings a number of fresh challenges, that the field of Service-Oriented Computing can help to address. Having a large number of autonomous and heterogeneous objects whose location, connectivity, and set of functionalities may change during a home’s life cycle, requires a rich and flexible infrastructure. Support for interoperation, dynamic discovery, sensing of the current execution context, and run-time service compositions are among the most notable elements of such an infrastructure.

In the context of the smart home the communication technologies of the devices differ on the physical transport level (e.g. network cables, power cable or wireless) as well as on the in the software protocols used for data transport (e.g. UPnP, X10 or Bluthooth). Archiving interoperability between devices using all these different methods of communication is no simple matter. Even if the challenges involving physical and data transport are tackled, the devices may use differ communication paradigms (e.g. event driven or service oriented). Supporting all of these communication paradigms and technologies in a uniform way is therefor a serious challenge.

In these thesis I propose to use an abstraction layer based on OSGi and UPnP to meet this challenge. This abstraction layer not only solves the communication difficulties between devices but also allows for easy simulation of devices using software. These simulated devices are very useful for testing and demonstration purposes.

Smart homes for All (SM4All) is a project is that focuses on creating a persons-centric middleware-platform for pervasive services (see Chapter 3 for more details). The project forms an excellent case study for a Service-Oriented middleware. With this in mind the following research question can be formulated:

*Can a Service-Oriented middleware be useful, in the context of the SM4All project, to implement a scalable system that allows interoperability between heterogeneous devices.*

In this thesis I will investigate this question. In order to evaluate the usefulness in the context of this question I have defined two criteria:

1. **The ability to visualize a simulated scenario, similar to the scenario described in [16].** The goal of this criteria is to establish if the middleware provides the necessary features to be useful in the context of the SM4All project. This criteria can be use to
evaluate if the functionality offered by the middleware is enough to be useful in the context of the SM4All project.

2. **Perform well for scenarios of the scale of the most complex SM4All scenarios.**

   The goal of this criteria is to measure if the performance and evaluate the scalability of the middleware. The performance should be good enough for every imaginable scenario in the scope of the SM4All project.

If both the functionality and the scalability are sufficient then the Service-Oriented middleware can be useful for the SM4All project.

In order too answer these questions some software is necessary. Firstly of course an implementation of a Service-Oriented middleware. Secondly to evaluate the simulated scenario a visualization tool and a planner are necessary. Lastly a automated testing framework to measure the performance of the Service-Oriented middleware.

The architecture of the Service-Oriented middleware I implemented is inspired by the pervasive layer described in [22]. The architecture of this pervasive layer addresses important issues such as device heterogeneity, scalability and context awareness by using a Service-Oriented middleware based on standard technologies such as OSGi and UPnP.

Implementing some really complex scenarios in order to evaluate the scalability is not feasible. I therefor implemented an stress-testing tool. This way it is not only possible to get hard timings, it is also possible to see trends when changing parameters.

The visualization is based on the Google Sketchup based approach suggested in [16]. The are some issues with this approach and its implementation: it does not scale well, it is polling based, adding new devices is complex. These issues can be resolved by making use of the Service-Oriented middleware that is inherently scalable, event based and easy to extend.

By removing thing unrelated form visualizing and handling them in the pervasive layer a these problems can be solved. This is in line with good software design principles such as single-responsibility, making the visualization tool simpler and easier to maintain.

In order to simulate scenarios a visualization tool alone is not enough as the systems also needs to know how to react to events. For this a planner is required. As a SM4All partner the Rug developed planner based on AI techniques. We integrated this planner with the pervasive layer to create an Service-Oriented domotic system. The this system is described in [15] and presented at the International Conference on Service Oriented Computing (ICSOC), which is a major conference in the field of Service-Oriented Computing. The paper [15] describes the system architecture and implementation, gives a good introduction to the workings of the planner, and evaluates the scalability and performance of the system.

In this thesis the focus lays more on the Service-Oriented middleware than on the planner. This means that this thesis goes into more detail about the implementation and architecture of the Service-Oriented middleware then the paper does but that it does not contain a description or evaluation of the planner.

The rest of this thesis will describe the process of designing, implementing and evaluating a Service-Oriented middleware. First in Chapter 2 an overview of related work is given. Then in Chapter 3 the SM4All project and its goals - which is a case study for the Service-Oriented middleware - are introduced. Then in Chapter 4 the architecture of the pervasive layer and the rationale behind it is explained. After that Chapter 5 describes the implementation of the
architecture. Then in Chapter 6 the performance of system is evaluated. Finally I reflect back on the goals and draw conclusions in Chapter 7.
Chapter 2

Related work

This Service-Oriented middleware this thesis presents touches the fields of pervasive systems, ubiquitous computing, service oriented computing (SOC) and spontaneous networking and takes place in the context of the SM4All project. In this chapter aims to give some background and context of these fields and the SM4All project.

Smart Homes 4 All (SM4All) is an European Union Seventh framework Specific Targeted Research Project (STREP) in the Objective ICT-2007.3.7: Networked Embedded and Control Systems. The goal of the project is that of designing and implementing an innovative middleware platform for inter-working of smart embedded services in immersive and person-centric environments, through the use of composability and semantic techniques for dynamic service reconfiguration [22]. Concretely this means that the SM4All system should allow persons easily to control the many devices in their home through a variety of interfaces. Expected applications vary form health-care scenarios where SM4All gives users independence to normal homes where SM4All can be used to increase the comfort and safety of users.

In [25] Weiser describes a vision of people and environments augmented with computational resources that provide information and services where and when desired. This introduced the notion of ubiquitous computing. Abowd and Mynatt [2] distinguish three interaction themes in which ubiquitous computing had developed: natural interfaces, context-aware applications, and automated capture and access. They argued that in the future an continuously available systems would play an important role in ubiquitous computing. In addition to that they argued that there where till important challenges on the topics of: presenting information at different levels of the periphery of human attention, connecting events in the physical and virtual worlds, modifying traditional HCI methods to support designing for informal, peripheral, and opportunistic behavior. The service oriented computing (SOC) paradigm uses services as main unit of abstraction. Services are autonomous, platform independent entities that who interact in a loosely coupled manner [20]. They can communicate and be described using XML based standards such as SOAP and WSDL. According to Papazoglou et al. [20] there are still significant research challenges on themes of service foundations, service composition, service management and monitoring, and service design and development.

There exist several event-based middleware systems, examples include Hermes [21] and BOSS [24]. The research on Hermes focuses on providing high scalability. This is less of a concern
the SM4All case because of the relatively small number of devices (ca. 100-1000) a house will contain. The BOSS project on the other hand focus more on smaller scale home environments that have more in common with the SM4All scenario. It also makes use of UPnP but focuses on sensors - which are basically just event sources - in stead of services which also can provide services to manipulate the environment.

Feeney et al. [11] examined five challenges in spontaneous networking: poorly defined network boundaries, the unplanned nature of the network, host that are not preconfigured and users are no experts. In the context of the SM4All project some of these challenges for the more generic care easier to overcome while other are more difficult. It is for example to preconfigure host to some extend because the context where they will be used is know in advanced, on the other hand the users may not only not be experts but the may also have a handicap or disability which can complicate things. These are thing that must be taken into account for the design of system.

Service Oriented Architectures have been widely proposed, e.g., UPnP or Jini [8]. A richer form of “pervasive SOA” is proposed in [17], where the importance for home networks with platform independence and loose coupling is advocated. In [17] the challenges that currently exist in interconnecting home devices are described, and it is recognized that OSGi can be useful for developing smart homes. In [18], the semantic annotation of OSGi description is proposed to improve the discovery process.

Alternatives for OSGi such as SENDA [13] have also been proposed. According to [13] SENDA focuses on minimalism, scalability and standard service and device interfaces. A SENDA prototype based on CORBA and some OMG standard services has been implemented, but a ready to use stable implementation was never released. OSGi on the other hand has multiple well tested implementations¹ that are used throughout many popular products².

Looking at UPnP [12], its use as low level home middleware has been often proposed, e.g., [24, 14]. There have been developed alternatives to UPnP most notably Devices Profile for Web Services (DPWS) [7]. The objectives of DPWS are similar to the objectives of UPnP but DPWS aims to use Web Service standards - WSDL 1.1, XML Schema, SOAP 1.2 and WS-* extensions(WS-Addressing, WS-MetadataExchange, WS-Transfer, WS-Policy, WS-Security, WS-Discovery and WS-Eventing) - to archive these goals. This approach certainly has advantages especially for interoperability with other systems based on Web Service standards. There are however also some disadvantages. One mayor disadvantage is the limited availability libraries and tooling support for the WS-* standards. An example of the use of DPWS in domotics is [26] which focuses on the design of a system which allows interoperability between Web Services and DPWS. DPWS is a logical choice in this case because of it offers easy interoperability with other Web Services.

In the case of SM4All better the tooling support for UPnP is an important factor. The inherit support for UPnP in OSGi is also a important advantage for UPnP in comparison with DPWS for SM4All.

²Eclipse(http://www.eclipse.org/), Sun Glassfish v3https://glassfish.dev.java.net/
Chapter 3

Case study

The Smart hoMes for All (SM4All) project\(^1\) is a EU funded project for which a consortium of academical and business partners work together. The SM4ALL project will investigate an innovative middleware platform for inter-working of smart embedded services in immersive and person-centric environments, through the use of composable and semantic techniques for dynamic service reconfiguration. It faces the challenging scenario of private houses and home-care assistance in presence of users with different abilities and needs (e.g., young able bodied, aged and disabled).

The specific composition of the Consortium, consisting of top-class universities and research centers (UOR, TUW, RUG, KTH and FOI), of user partners specialized in domotics and home-care assistance (FSL and THFL) and a SME specialized in specific brain-computer interfaces (GTEC), and of leader companies in the embedded sector (TID and ED) guarantees a widespread dissemination and exploitation of the project results, coupled with a privileged position inside ARTEMIS\(^2\) and ARTEMISIA (due to the presence of UOR, TUW and ED in such bodies) [1].

3.1 SM4All Motivation and Objective

Embedded systems are specialized computers used in larger systems or machines to control equipments such as automobiles, home appliances, communication, control and office machines. The fact that such systems are everywhere and in our everyday life has been well recognized by different European actors. Such pervasivity is particularly evident in immersive realities, i.e., scenarios in which invisible embedded systems need to continuously interact with human users, in order to provide continuous sensed information and to react to service requests from the users themselves. Examples of such scenarios are digital libraries and eTourism, automotive, next generation buildings and infrastructures, eHealth, domotics. Having the users at the centre poses many new challenges to the current middleware and service technologies for embedded systems, in terms of

\(^1\)http://www.sm4all-project.eu/

\(^2\)http://www.artemis-office.org
• **Dynamicity:** sensors and services are no more static, as in classical networks, e.g., for environmental monitoring and management or surveillance, but the overall distributed system (consisting of all the sensors and devices and appliances) need to continuously adapt on the basis of the user context, habits, etc., by adding/removing/composing on-the-fly basic elements (services offered by sensors/devices/appliances);

• **Scalability:** in order to really immerse the users in the system, the number of sensors/devices/appliances should be huge, at least an order of magnitude more than the current situations. As an example, the current best-in-class smart houses count for tenths of sensors/devices/appliances, the next generation smart houses for all will count hundreds of devices;

• **Dependability:** when the users are at the centre, they heavily depend and thrust on the environment/system itself, and therefore it should be highly dependable;

• **Security and privacy:** when the users are at the centre, the security of the overall invisible environment is crucial; moreover, such an environment, if hacked, could potentially provide any sensible information on the users (this is especially critical when the users have some disease, disabilities, etc.), and therefore the design of such a system should pay specific attention to privacy preservation, that should be built-in in the system, and not added-on later, as in current design practices.

These considerations require, in the Consortium opinion, novel techniques and middleware technologies targeted to person-centric embedded systems (i.e., usable in immersive scenarios) [1].

### 3.2 The SM4ALL Vision and Concepts

The SM4ALL (Smart hoMes for All) project aims at studying and developing an innovative middleware platform for inter-working of smart embedded services 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 private/home/building in presence of users with different abilities and needs (e.g., young able bodied aged and disabled).

In particular, in the SM4ALL project P2P, service-orientation and context-awareness are merged in novel ways in order to define a general reference architecture for embedded middleware targeted to immersive scenarios, among which the domotics and home-care have been selected as showcases. The SM4ALL platform will offer specific features of scalability (with respect to the number of embedded services to be managed) and dynamicity (the capability of the platform to manage on-the-fly insertion/removal of (new) devices/sensors/appliances). By exploiting techniques from autonomic computing (self-* properties) and peer-to-peer systems, the services and the platform will offer such features directly built-in, and not as a subsequent add-on. Moreover, in the design of the SM4ALL platform, there will be a specific focus on ontologies for describing service capabilities, to be used for obtaining the dynamic configuration and composition of the services, while preserving the privacy of the users. Specific emphasis is devoted to the overall security of the smart environment, with respect to who/which can do what and to possible intrusions of spy sensors/services/devices.
Finally the SM4ALL project aims at defining and developing such an embedded pervasive platform for smart houses truly for all, in which users with different abilities and needs (e.g., young able bodied, aged and disabled) can interact with the services provided by the different domotic devices, appliances and sensors through basic and advanced interfaces, being the latter ones based on brain-computer interaction technologies. Brain-computer interaction (BCI) is a specific set of techniques, based on the interplay of hardware and software, that allows people to interact with a screen; in the project vision, it allows the selection of a desired goal among a set of possible ones, proactively offered by the SM4ALL system on the basis of the available services, the current context of the user (perceived through the sensors and a profiling of previous actions and goals). Once the desired goal is specified, some composition techniques will define the most suitable way of coordinating the available services, and will deploy such an orchestration specification on top of the infrastructure, through which the services interact each other in order to deliver some final composite service to the user. Such a composite service can effectively satisfy the user’s goal, or can take the infrastructure “nearer” to it than in the initial situation, not being able to progress any more if not correctly driven by the user. Again the SM4ALL system propose new possible alternatives, and the cycle (user makes a choice, composition and orchestration) is repeated until the user’s satisfaction is reached. In such a way, the continuous interaction between the user and the house infrastructure (that acts proactively and automatically among different interaction points, and it is not “passive” as current houses are) allows the user to satisfy his needs by relieving him from low level coordination of devices’ tasks. This is especially useful in case of permanent, partial or temporary disabilities, as the SM4ALL showcases will demonstrate.

In this way, the ambitious target of the SM4ALL project is to couple, for the first time, advanced research and techniques in embedded and distributed systems with service oriented computing (on the one hand) and accessibility and advanced interaction techniques (on the other hand), thus developing a truly embedded dynamic environment for all [1].

3.3 Scenarios

In [19] two showcases that give a good indication of the advantages of the SM4All are presented. The first one involves Nils, a 34 year old which is affected by Amyotrophic Lateral Sclerosis (ALS). The second one involves Mike who hosts a weekly gathering with some friends. Those two showcases highlight the diversity of the scenarios that SM4All should enable. For Nils the system means a greater amount of independence and freedom while for Mike it is just a quick way to prepare for some friends coming over. Form these showcases we will highlight what SM4All offers to the actors.

3.3.1 Showcase Nils

Nils is 34 years old, and is affected by Amyotrophic Lateral Sclerosis (ALS). ALS is a progressive, fatal, neurodegenerative disease caused by the degeneration of motor neurons, the nerve cells in the central nervous system that control voluntary muscle movements. This disease is progressive, and it is probable that Nils in the future will not be able to make any muscle movement.

At present Nils can move his arms, head and upper body, but he cannot move any other part of his body. With some difficulty he can leave his bed and sit on a wheelchair, but he needs the help of someone in order to do it. This fact forces Nils to stay in bed for long time, waiting for
someone who can help him to change position and to go out of the bedroom for eating or caring his personal hygiene.

In order to help Nils when no one else is at home, the whole house is managed by an automatic home system controlling most of the devices in it. The home system can be driven by Nils as well, by means of a Brain Computer Interface (BCI) or through a keyboard. He prefers using the BCI system for controlling his surroundings, because in the future BCI will be his only way for communicating, and he wants to practice with it.

Using the BCI Nils can check the supplies he has in his fridges and order things over the internet if necessary. It also possible for Nils to control multiple devices by a single action. If for example Nils uses the BCI to indicate he wants to sleep, all devices in the house are put in the optimal state for this. This means that the lights turn off, the TV goes on standby, the bed goes in optimal mode and his alarm clock asks if he wants to wake up at a certain time using data from his agenda.

3.3.2 Showcase Mike

Mike lives in a home with his sister Maria. On Friday evening Mike has some of friends come over to his place. He can remotely access his video collection using the internet. When Mike discovers he has no suitable films for the evening he can use automatically send out a poll per SMS to his friends based on films suggested to him based on his preferences. Mike can then start the download the film to his home, which will be automatically paid using his payment gateway.

In his house the SM4All system can detect fires. When that happens it automatically opens and closes doors and windows depending on the location of the fire and the location of the actors, in order to contain and extinguish the fire.

3.3.3 Scenario analysis

The scenario’s of Nils and Mark user respectively 25 and 34 devices. This does not include the devices that people bring with when the visit, such as smart phones. Taking into account that both Mark and Nils live in relatively small homes the number of devices can increase even more in bigger homes. This leads to the estimation that in a most scenario’s the total device count will lower then hundred, an realistic upper bound for big homes with lots of people visiting seems to be a couple hundred devices (300-500).

As the showcases demonstrate quick response times are important for both the usability and safety of the system. The middleware is responsible for the communication between devices. This means that in order for the system to be fast the middleware should be fast. This results in an interesting challenge: a few hundred devices using a wide array of different communication technologies must be able to communicate quickly with each other.
3.4 SM4All architecture

3.4.1 General architecture

The general architecture of the SM4All project is described in [22]. The SM4All consortium is especially values the following system characteristics:

- **Person centric**: The system should above all else be focus on providing a good experience for the users.

- **Globally distributed, communication and service centric**: The system should exists of distributed services can seamlessly interact with each other.

- **Openness and reuse**: The system should focus on open standards to encourage reuse and allow easy interoperability and extensibility.

It is important that the system has these characteristics, Figure 3.1 shows an overview the SM4All architecture which has these characteristics.

![Figure 3.1: SM4ALL architecture overview (source: D1.1 Project Presentation[23])](image)

12
The main components of this architecture are:

- **Sensors and devices.** Hardware to gather data in interact with the world e.g. smoke sensors, smartphones with a ui for the system, motors to open a door.

- **Embedded services.** Both sensors and devices can expose their functionalities as services. This can happen in both a synchronous and asynchronous(event driven) manner. These services are the cornerstone of the Service Oriented Architecture (SOA) of SM4All.

- **Data distribution bus.** Devices/appliances/sensors using a wide spectrum of networking technologies can inter-operate using the data distribution bus.

- **Embedded distributed orchestration engine.** Coordination of services can be carried out by a specific orchestration component.

- **Composition engine.** When the user selects a desired goal (e.g. watch TV, take a bath) the composition engine will synthesize the right orchestration of services to able to satisfy such a goal.

- **Repository of service description.** Service descriptions will to be stored on a repository, such that they can be used during the synthesis process.

- **Context-aware user profiler.** The SM4ALL system needs to continuously profile the user, in order to proactively propose goals to him and to depict a clear picture of all the states he may go throughout.

- **User interface.** The user is able to interact with the house through many interfaces such as a home control station, a smartphone or a Brain Computer Interface(BCI).

### 3.4.2 Pervasive layer architecture

For the context as of this thesis the pervasive layer is especially interesting as it deals with eventing and service oriented networking. Figure 3.2 gives an overview of the network architecture of SM4All. In the this architecture the SM4All middleware coordinates devices using various networking protocols to from a smart home. This smart home is exposed to the Internet via a home gateway.
To enable communication between devices using a wide spectrum of networking technologies - such as UPnP, X10, Bluethooth, ZigBee - an UPnP abstraction layer is introduced. This means that all communication is translated first to UPnP before it is processed by the system. Information coming from the middleware is translated the native format of a device before it send to the device. The UPnP protocol is used as abstraction because it supports all concepts that the SM4All system needs in order to function. This means it is always possible to translate a networking technology to UPnP without losing functionality that is useful to the system.

The home gateway is the core element of a home, it coordinates the internal communication between devices and provides means to interact with the outside world. In order to do this the home gateway should provide the following functionality:

- Device remote management and interoperability.
- Dynamic management of services and devices, providing automatic deployment, execution, maintenance and installation.
- QoS provision and management for operator’s services.
- Single control point for devices and services based on UPnP.
- Secure management of services and user information.
- Support for integration for different communication technologies in the home environment such as Ethernet, Wifi, PLC, Zigbee or Bluetooth.
Chapter 4

Architecture

This chapter describes the architecture of the pervasive layer. First the most important requirements for the pervasive layer are introduced. Next is an overview of the architecture meets these requirements. Next the rational behind the architecture is explained. Finally most important components of the architecture are described in more detail.

4.1 Goals/Requirements

The general system characteristics described in 3.4 and functionality of the home gateway described in section 3.4.2 contain the most important requirements for the pervasive layer. Based on these sources one could formulate the goal of the pervasive layer to be: provide a dynamic, transparent, robust and scalable service oriented platform for event driven service communication. In terms of concrete requirements the following are especially important:

1. **Service discovery.** The system must detect when new services become available and notifying interested parties.

2. **Service description.** The services should be described in a standardized programmatic interpretable manner.

3. **Services control.** It should be possible to control services using the operations they support.

4. **Eventing.** Interested parties should be notified of state changes of services in a event driven manner.

5. **Platform independence.** The services should communicate via standards in order to enable communication between services regardless of the platform they run on.

6. **Extensibility.** It should be easy to extend the platform. This means is should be easy to write new services for the platform. It also means that the architecture should support future requirements that are likely to arise such as security and concurrency management.
7. **Open standards and reuse.** The system should use open standards as much as possible in order to stimulate a dynamic, heterogeneous environment in which devices and services can interact. The when suited open source software is available that should be used to prevent reinventing the wheel.

4.2 **Architectural overview**

In order meet the requirements an Service Oriented Architecture (SOA) is used. An overview of this architecture is shown in Figure 4.1.

![Figure 4.1: Architectural overview](image_url)

In the architecture of three main components - pervasive clients, a pervasive middleware and a device layer - can be distinguished.

**The device layer** is where the physical devices are located. For UPnP devices will use TCP/IP and UDP as basic networking protocols to communicate with the pervasive middleware. But it is also to use other networking for devices to use other technologies such as Bluetooth or ZigBee.

**The pervasive middleware** contains the logic to manage and control the devices and support
eventing and expose them to pervasive clients. The pervasive layers contains a *device abstraction layer* which abstracts away the underlying device technology. For every technology the system supports there needs to be a driver. The responsibility of a driver is to wrap devices as instances of the UPnPDevice interface that and register them as a OSGi service with the framework. In this way they can discovered by other OSGi bundles such as the *home controller*.

**The pervasive clients** contains pervasive clients that use SOAP to communicate with the pervasive middleware. Currently there are two clients implemented, a planner and a visualization client. The planner creates and executes plans which control the services in the house based on the current state of the services. The visualization component provides a 3D visualization of the current state of the house. Both client are independent of each other they only communicate with the middleware.

In each of these components services play a role often on a different level of abstraction.

At the lowest level at the device layer service are exposed by devices. Each device expose its functionality through one or more services. The technology used to describe and interact with this services does not matter as long as there is a driver available for it in the pervasive middleware.

In the device abstraction layer of the pervasive middleware devices are abstracted to OSGi services. This means that in the OSGi framework a device is represented as OSGi service which exposes the services the device offers through its interface\(^1\). The home gateway in its turn exposes a web service interface to manage and interact with the OSGi service (which represent devices).

### 4.3 Design rationale and alternatives

In this section an overview we try to explain rationale behind the architecture and we discuss what alternative options we considered. First we have a look at the technologies we have chosen to use. In the second part we explain the roles of different design patterns in the architecture.

#### 4.3.1 Technologies

The OSGi framework is a industry standard with multiple stable implementations. Given the requirements of open standard and reuse and extensibility combined with the fact that the OSGi is service oriented, make OSGi a very useful technology for the pervasive layer.

The requirement device independence implies the use of an device abstraction layer. This means it must be represent abstract devices and service. When taking into account the reuse and open standard requirement two technologies seems to be suited DPWS and UPnP. The main advantage of DPWS over UPnP is that it is completely build of WS standards and extensions where UPnP uses some non WS standards. Unfortunately the tooling support for most of the WS extensions used by DPWS is really lacking. UPnP on the other and is supported by the OSGi framework by default which makes things a lot easier. Therefor UPnP is used as device neutral technology by the device abstraction layer.

In order to expose the house to the internet a home gateway is used as central access point.

\(^1\)org.osgi.service.upnp.UPnPDevice (see [http://www.osgi.org/javadoc/r4v42/org/osgi/service/upnp/UPnPDevice.html](http://www.osgi.org/javadoc/r4v42/org/osgi/service/upnp/UPnPDevice.html))
An alternative to expose every service directly to the internet. Both approaches have their advantages.

In a scenario where each service is accessible for the directly internet it would still be necessary to have central point access url’s of the active services and subscribe to events for when services become unavailable. The service description, control and service state related eventing could then be handled by the service exposed to the internet. While solution this may be cleaner in the architectural sense it has some disadvantages. The first one is complexity because a central service with some functionality needs to be present anyway it is simpler to extend this service with some operations that apply to individual services then to create a separate web service for each service. The second one is that easier to improve performance of a central service because it is possible to commands for multiple service over the same http connection in order lower the number of http connections that has to be made. This is impossible to do when each service is available as a web service as you need to make a separate connection to each service. For these reasons we have chosen an architecture with a central point access.

For communication with the outside world there are we considered two styles that are used on the modern web the: Rest style and the RPC style. There Rest style involves the use of resource identifiers and http verbs (GET, PUT, POST, DELETE) to create web services. This style is very useful when the operations are easily mapped to http verbs. In the case of the home gateway this mapping would not be really intuitive, therefore we choose to use a RPC style solution. Two RPC style technologies are SOAP and XML-RPC. The advantage of XML-RPC is that it is simpler that SOAP. A disadvantage of XML-RPC is that it lacks a service description language where SOAP has WSDL to as description language. As a we find a clear definition of home gateway important we choose to use SOAP+WSDL over XML-RPC.

4.3.2 Design patterns

Design patterns are general reusable solutions to a commonly occurring problems in software design. The exact implementation of a pattern can vary per occasion but they share an common structure that can be reused. Using design patterns has several advantages such as faster development and improved maintainability. Patterns can be categorized based on the similarities in structure or the type problem they solve. In the pervasive layer eventing is an important challenge, to overcome this challenge we use publish subscribe based patterns. An other important challenge is device and platform independence, which basically means that device and platform specific details need to be abstracted away. To archive this the layers pattern is used.

Publish/subscribe

The 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]. Decoupling it space means that the parties - publishers, subscribes - neither know each other nor know many other parties there are. Decoupling in time means that the parties do not have to be active at the same time. Decoupling in synchronization means that is possible to generate and process events both synchronous and asynchronous. There are a lot of

18
publish/subscribe based patterns, all offering specific advantages. Below two variations that are used in architecture of the pervasive layer are described.

**Listener**

The listener pattern is a simple pattern for publish/subscribe based eventing. Event sinks register themselves with a event source. When an event occurs the event source notifies the event sinks. Figure 4.2 show a sequence diagram of the interaction between event sources and event sinks.

![Sequence diagram of listener pattern](image)

Figure 4.2: Sequence diagram of listener pattern

The listener pattern is used throughout the Java standard library. The event driven communication between the pervasive server and the pervasive clients is also based on the listener pattern. In this case the communication uses SOAP and the interfaces are defined in WSDL instead of Java but the pattern stays the same. The registration of listeners happens via the registerClient and subscribe methods. The client callbacks - serviceDiscovered, serviceUnavailable, stateVariableChanged - are implement as webservice operations.

**Whiteboard**

The whiteboard pattern is the eventing mechanism of the OSGi framework[3]. In the whiteboard pattern event sinks register with a registry. Event sources use the registry to look up event sinks when events occur. Figure 4.3 shows the interaction between sources and sinks in the whiteboard pattern. The use of a central registry simplifies the registration for sinks because they only have to register in one place and do not need to know all event sources in the system. Event sources can use the registry to easily lookup sinks. The use of an filter makes it possible the receive only the events in which sinks are interested minimizing the overhead.
In the pervasive layer the whiteboard pattern is used for the event based communication between the UPnP importer/exporter and the controller.

Layers

The layers pattern is a pattern to deal with different levels of abstraction within a system. In the layers pattern components are only available in one layer. Layers can only communicate with the layer one level below or above itself. This way the layers pattern can hide away implementation details in lower levels. This makes it an good pattern archive device independence as device specific details can be hidden behind the device abstraction layer.

4.4 Eventing

An important feature of the pervasive layer is its support for event driven communication. Communication in an event driven manner is a lot more efficient the communication based on polling. The eventing support of pervasive layer is base on the UPnP eventing mechanism. This is because all devices are abstracted to UPnP in the device abstraction layer. UPnP eventing is based on so called state-variables which represent a (part of) the visible state of a UPnP service. Using a publish/subscribe based technology interested parties can register to such state-variables. If the such a state variable then changes the registered parties receive a notification.
Figure 4.4 shows the flow of an event which start as an UPnP state-variable that is changed passes through the layers and eventually reaches the web services clients. As soon as a 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 [3].
3. The UPnP service wrapper propagates the event to the Controller.
4. The Controller running in the OSGi framework notifies the Server which as a separate process outside the OSGi framework.
5. The Server invokes the stateVariableChanged operation (defined in the clients WSDL) for registered clients.

4.5 Technologies

There are two technologies that are especially important for the pervasive layer: UPnP and OSGi. In the following sections these technologies and how they are used will briefly be described.
4.5.1 UPnP

The Universal Plug and Play (UPnP) standard defines a set of networking protocols based on established standards such as TCP/IP, UDP, HTTP, XML, and SOAP. The UPnP protocol has support for discovery, description, control, event notification and presentation [12]. UPnP uses Simple Service Discovery Protocol (SSDP) for discovery, which uses UDP unicast and multicast to discover services. 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 describe the operations a service supports they may have input and output arguments. Actions are used to control UPnP services. State variables describe the current state of an UPnP service, 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). GENA uses HTTP like request with a XML message body. UPnP archives platform independence because it is build upon standard technologies such as TCP/IP, UDP, HTTP, XML, and SOAP.

4.5.2 OSGi

The OSGi framework is a dynamic module system and service platform for Java[4]. It provides additional features that the Java Platform does not provide by default such as life cycle management for services and dependency management. OSGi program are distributed as so called bundles. These OSGi bundles are Java .jar files with extra meta-data added.

The OSGi framework also provides an Java interfaces to define and interact with UPnP devices. Implementing these interfaces is far simpler than implementing UPnP devices using the lower level standards of UPnP directly.

To enable non UPnP devices to be managed by the controller a UPnP proxy needs to created. Because the OSGi framework simplifies implementing a UPnP service greatly, developing a proxy for non UPnP hardware should generally take not to much effort. This allows for faster development of services.

The flexibility of the OSGi framework the architecture helps to make the architecture extensible. It is not a new idea use OSGi to add extensibility, the Java IDE Eclipse2 for example uses OSGi for its plugin architecture and the new Sun Glassfish v33 Java EE server also uses OSGi to add extensibility. Adding functionality such as logging or monitoring can be easily implemented in a modular fashion using OSGi bundles.

With regard to eventing the OSGi framework uses the whiteboard pattern[3]. In the whiteboard pattern the event source is not registered as a service, instead event listeners are registered are registered. If a event occurs in a event source it looks up the registered listeners and sends the event to them.

\[http://www.eclipse.org/\]

\[urlhttps://glassfish.dev.java.net/\]
4.6 Main components

The main components of the architecture are introduced in Section 4.2. There are three main components that can be distinguished: the home controller, the home gateway and the drivers. The drivers operate as bridge between the hardware and the pervasive middleware. The home gateway and the drivers both live in an OSGi environment. The home controller makes the necessary functionality from within the OSGi framework to the home gateway. The home gateway is a Web-Service that clients can use to interact with the pervasive middleware.

The next few sections will explain the workings of these components in more detail.

4.6.1 Home controller

The home controller is a OSGi bundle that is responsible for handling eventing and control of the UPnP services available in the OSGi framework. It functions as a bridge between the OSGi layer and the web home gateway.

4.6.2 Home gateway

The home gateway is a Web Service which allows clients to control services and receive notifications of events. It exposes the following operations to clients through a WSDL interface:

1. **(de)registerClient**, (de)registering a client. Clients are identified be the url of the client service wsdl file which should be passed as an argument. Once a client is registered it receives event notifications when UPnP services become available and unavailable.

2. **(un)subscribe**, (un)subscribing to state-variables of a services. Once a client is registered to a state-variable it receives event notifications whenever the state-variable is changed.

3. **listServices**, returns a list of service descriptions - including service properties, actions and state-variables - of all currently discovered services.

4. **invokeAction**, invokes a specified action from a service with the given arguments and returns the values of output arguments.

**Client web services**

Clients are also implemented as web services. It exposes the following the following callback operations:

1. **service(un)Available**, once a client is registered the appropriate operation is called whenever a service becomes available or unavailable.

2. **stateVariableChanged**, if a state-variable to which a client is subscribed changed the operation is called.
4.6.3 Drivers

Drivers take care of detecting new devices on the home network and creating an appropriate OSGi UPnP device proxy. This abstracts away the devices specific technologies and allows the components further up in the stack to communicate in a unified way with all devices. The proxy should implement the UPnPDevice interface defined by the OSGi specification and be registered properly with the OSGi framework. If the driver takes care of that device is automatically picked up by the home controller and can directly be used.

It is also possible create so called simulated devices by implementing the UPnPDevice by custom logic and registering it with the OSGi framework. This can be very useful for testing and simulations but it can also be useful as flexible mechanism to provided additional software services.
Chapter 5

Implementation

The Sm4all project is currently underway and various implementation efforts are being carried out by the partners. Currently work is done on important components of the system such as the composition layer and the brain computer interface. An implementation effort especially worth mentioning is [16] as was an primary inspiration for this implementation of the pervasive layer. The implementation of the pervasive layer described in this thesis is a working and completely functional prototype.

This chapter has two main goals, the first is to describe how the architecture is implemented the second is to provide information on how to build upon and extend the pervasive layer. Section 5.1 describes the implementation of the basic components of the pervasive layer. Section 5.2 describes the extension mechanism’s of the pervasive layer and the communication between the home gateway an pervasive clients.

5.1 Pervasive layer components

5.1.1 Home gateway

The home gateway is written in Java. It uses the Apache CXF framework\(^1\) to run the web service and to generate Java from the WSDL files.

The home gateway is the “glue” between pervasive clients and the pervasive middleware. It handles the communication with the controller, keeps track of registered pervasive clients and propagates events form the home controller to the registered clients. The communication with the home controller happens via message passing over a socket.

Clients can register themselves with the pervasive server by invoking the registerClient operation with url of their WSDL file as argument. The WSDL file needs to the same as the client WSDL contract with the exception of location of the port which must be update to point to the location where the client webservice is running. The pervasive server will use this WSDL to generate a

\(^1\)http://cxf.apache.org/
proxy and store it in the registry. As long as the client stay registered this proxy is used to notify
the client of the events it is subscribed to.

Registered clients will be notified of when services become (un)available. Clients can also sub-
scribe to state-variables of UPnP services.

5.1.2 OSGi framework

The current version the pervasive layer runs on the Apache Felix\textsuperscript{2} implementation of the OSGi
framework. The components of the pervasive layer do not use implementation specific func-
tionality i.e. only functionality that is specified by the OSGi standard. This means it should
be relatively easy to switch to another implementation of the OSGi v4.2 specification such as
Knopflerfish\textsuperscript{3} or Equinox\textsuperscript{4}. The system works fine with Apache Felix so there is currently no
need to switch to another OSGi implementation.

5.1.3 Home controller

The controller is implemented as a OSGi bundle. It uses message passing over a socket to
communicate with the pervasive server. This has as advantage that the pervasive server and
the controller can run on different machines. Furthermore it has the advantage 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 that is used in the OSGi framework and the eventing based on the listener
pattern that takes places in the home gateway.

5.1.4 Drivers

For the importing and exporting of UPnP devices a OSGi bundle provided with Apache Felix is
used. This bundle provides all the necessary functionality to import and export UPnP devices
to the OSGi framework. The UPnP driver is currently that is the only driver that is available.
Given the good support for UPnP by OSGi and the availability of the UPnP driver as example
it should not be too hard to implement drivers for other technologies.

5.1.5 Virtual devices

Virtual devices (also know as simulated devices) are devices that have are implemented only in
software an have no real hardware. Virtual devices can be useful for testing and demonstration
purposes. Virtual devices 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 are function as real
devices in the system.

\textsuperscript{2}http://felix.apache.org/
\textsuperscript{3}http://www.knopflerfish.org/
\textsuperscript{4}http://www.eclipse.org/equinox/
5.2 Extending and using the pervasive layer

There are two basic ways to extend the pervasive layer, the first is to implement a pervasive client the second is to implement a OSGi bundle. Table 5.1 shows the advantage and disadvantages of using OSGi bundles or pervasive clients to extend the system.

<table>
<thead>
<tr>
<th>General properties</th>
<th>OSGi bundle</th>
<th>pervasive client</th>
</tr>
</thead>
<tbody>
<tr>
<td>distributed</td>
<td>- no, runs in OSGi process</td>
<td>+ yes, can run on different machine across network</td>
</tr>
<tr>
<td></td>
<td>+ fast, Java in-process call</td>
<td>- slow, SOAP calls</td>
</tr>
<tr>
<td>computation</td>
<td>- not scalable, runs in OSGi process</td>
<td>+ scalable, computations can be performed distributed</td>
</tr>
<tr>
<td>performance</td>
<td>- must be Java</td>
<td>+ any language</td>
</tr>
<tr>
<td>language</td>
<td>+ full OSGi &amp; Home controller API</td>
<td>- more limited home gateway API</td>
</tr>
<tr>
<td>API functionality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy of implement-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dependencies</td>
<td>- must be made explicit</td>
<td>+ doesn’t matter</td>
</tr>
<tr>
<td>eventing</td>
<td>+ simple using whiteboard pattern</td>
<td>- more complex, must implement callback webservice</td>
</tr>
<tr>
<td>API complexity</td>
<td>- more complex OSGi &amp; Home controller API</td>
<td>+ simpler home gateway API</td>
</tr>
</tbody>
</table>

Table 5.1: Differences between OSGi bundles and pervasive clients as extensions

The biggest difference between a OSGi bundle and a pervasive client is that OSGi bundles run in the same process as the OSGi framework where pervasive clients can run on a other machine across a network. Most other properties - performance, complexity of eventing, etc. - are the result of this difference.

Currently there are two extensions implemented for the pervasive layer, a visualizer and a planner. Both are implemented as pervasive clients but for different reasons. The visualizer is implemented as pervasive client because the visualization must be available over a network and not just at the computer the runs the pervasive layer. The planner as is implemented because it is more CPU intensive than IO intensive, meaning that the loss of performance by using SOAP calls (IO) is more than made up for by the fact that the computation of a plan (which is CPU bound) goes faster and doesn’t slow the rest of the system down. In the sections 5.2.3, 5.2.3 the a brief description of the implementation of these clients is given.

Describing the development of OSGi bundle extensions goes beyond the scope of the thesis, for those interested more information about OSGi development can be found at [http://www.osgi.org](http://www.osgi.org). Before going into the details on how implement a client it is good to have look at the contract first methodology the is used to generate communication related code, this is done in section 5.2.1. After that the interaction between client and server is described, which should be useful as a guideline on how to implement proper behaving clients.
5.2.1 Contract first service development

The web services are developed in a contract first manner, meaning a WSDL file is written first and code is generated from this WSDL. The alternative is code first, which means code is written first and the WSDL file from the code. Contract first maintains many of the inherit advantages of SOA - interoperability, robustness, extensibility - better than code first[6, 5].

Interoperability is especially important because the pervasive server is written in Java while the visualization tool is written in Ruby. To support eventing there needs to be two-way communication between the pervasive server and the pervasive clients. This means that a Ruby client must be able to interact with a Java server but also the other way around. The language constructs of general-purpose programming languages that tools use to generate WSDL are not as expressive as WSDL constructs when it comes to describing services. This means that to control the details of WSDL, the tools need additional metadata. But even if the tool supports additional some for metadata it is often impossible to control the finest details of the generated WSDL. The tooling support for WSDL for some languages such as Ruby is less than optimal. These factors combined make contract first development the better solution in this case.

The code generation scheme for two-way communication using contract first development is show in Figure 5.1. There are two services described by WSDL files with share their data types defined in 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 the operations to register, control and subscribe to services. A client proxy is instantiated for each client that registers and used for further communication with that client.

For the “client side” (pervasive clients) a stub is generated from the client WSDL and a proxy for the server WSDL. The stub is used to implement the logic for handling events form the pervasive layer. The proxy is used to register, control and subscribe to services in the pervasive layer.

5.2.2 Client server interaction

There are two ways a client can interact with the pervasive server either with or without event support.
Figure 5.2: Class diagram of the Java classes generated of the common types XSD
The simplest way to interact with the server is without event support as no initialization is required. A client can just start invoking operations directly. The only operations that can be used in a scenario without events are listServices and invokeAction. Still these operations are enough to control the system. It is possible to get the current state of the system has listServices and a number of (one per state-variable getter) invokeAction calls and control the system using invokeAction calls. If it is necessary to frequently check for changes in the state of the system the repetitive calls to listServices and invokeAction can quickly become a performance bottleneck. ListServices is a relatively slow operation as it returns a description of all active services (which can be quite big if there are a lot of services active) and invokeAction needs to be invoked for every state-variable (which will become slow when there are a lot of state variables. In a scenario where a client-side up to date representation of the system is required not using events therefore clearly is a bad choice, and an eventing should be used.

The disadvantage of using eventing is that it is more complex to implement a client for it. This extra complexity is caused by two things: the client has to implement a webservice to handle events, and there is some initialization necessary before the eventing can take place. The webservice the client needs to implement is described by a WSDL file an tools can be used to generate a stub for it see section 5.2.1. Because implementers are likely to used a tool to generate a proxy for the server webservice in anyway, this should not cost too much effort.

In the initialization phase the client should first start its callback service. Then the client can invoke the registerClient with the url to the callback service as arguments. This cause the client to receive notifications as new services are discovered and when services become unavailable. The next step is subscription to all the state-variables of services in which the client is interested. In order to do this the client can invoke the listServices operation of the server which will return a list of descriptions of the currently active services. From this list the client can then select the state-variables of services in which the client is interested and call the subscribe operations with a service name and a list of state-variable names to subscribe to. Note that this is the same procedure that non eventing clients can use to get the system state, in an evening scenario however this takes only once - during the initialization - and not every time a client wants to have a up to date view of the state of the system. Once this operation is completed the client will receive events a state-variable changes. Figure 5.3 shows a sequence diagram of this process.

Figure 5.3: Sequence diagram of client initialization
After a client has initialized properly it starts receiving notifications of events when they occur as illustrated in the sequence diagram Figure 5.4. The serviceDiscovered and serviceUnavailable events probably only be used to update the subscriptions i.e. once a new service is discovered the client checks if is interested in this service and if so invokes the subscribe operation of the server for this operation. Once a state-variable is changed the client will likely perform its real task such as updating visualization (for the visualizer) or generating a new plan (for the planner).

![Sequence diagram of client initialization](image)

Figure 5.4: Sequence diagram of client initialization

### 5.2.3 Client implementations

#### Visualizer

The Visualizer visualizes the current state of the home using Google SketchUp. It is based upon the visualizer described in [16], improving it by adding support for eventing and pushing down device implementation details to the pervasive layer. The visual model of the house updated in a event driven manner using the Ruby API of SketchUp. The planner is an example of the platform independence offered by the pervasive layer offers. The planner is implemented in Ruby communicates with the pervasive layer written in Java using web-services defined by WSDL.
Because Google SketchUp does not allow to run background threads the client web service runs in a separate process, where it queues events it receives from the pervasive layer. A script inside SketchUp uses RPC to poll the event-queue every 100ms using a timer which is included in the SketchUp API. Figure 5.5 shows this event handling process.

**Planner**

The planner is the module standing at the core of the composition layer. Its task is to compute a plan, that is, a sequence of actions that need to be applied in order to satisfy a given goal. Starting from an initial state, which in our case is reflected by the current values of all variables that describe the home domain, the application of each action in the plan leads to a new state, as prescribed by its effects. The planner first retrieves the description of the home stored in the repository, and forms the planning domain, by mapping each UPnP action to a planning-level action, specified in terms of preconditions and effects. This process has to take place once for each house instance, and be repeated only when a new service is discovered or removed.
Chapter 6

Evaluation

It is important to evaluate the performance middleware systems such as the one presented in this thesis. Because middleware systems connect different software components together they can drag down performance of an entire system if they do not perform well.

In the context the SM4All project it is important that the system can handle the amount of devices which will be present in the most ambitious scenarios well. Furthermore it has also to be taken into account the more clients will be added to the system in order to extend the functionality of the system. In this chapter we will evaluate how the system performs with an increasing number of clients and how it scales with an increasing amount of devices.

6.1 Performance test

A first indication that the system perform well in real world scenarios is the performance of the visualization tool, which runs stable and seems to react near instantaneous to events of both real and simulated devices. This inspired confidence but provided also raised the question how fast the system really is and how well it performs when the number of clients and services increases. In order to get answers to these questions it is necessary to perform automated performance tests.

6.1.1 Latency

Latency is an important measure for the performance of interactive systems [9]. Because the pervasive layer is a highly interactive system latency is an important measure of the performance of the pervasive layer.

Latency is the absolute temporal difference between stimulus and response to an action.

For the purpose of the evaluation the following quantities and measures are defined: $t_{stimulus}$ is the time at the beginning of the invocation of a UPnP action of one of the test devices, and $t_{response}$ the time a client receives a notification that a state variable changed as result of the invocation of an action. Using those measures the latency can be defined as $t_{latency} =
With the choice of $t_{\text{stimulus}}$ and $t_{\text{response}}$ as described the latency covers both control and eventing (as invoking a action is control and receiving the event is eventing).

### 6.1.2 Test method

First, we test the latency in the pervasive layer. The setup is based on using a 2.66 Ghz computer running Windows 7, 64 bit and Java 1.6.0_18. The devices used for the test are implemented as OSGi bundles, i.e., the devices are simulated and wrapped in OSGi. Every device has one service which has a state variable and action. The clients are implemented in Java using Apache CXF 2.2.5. For the purposes of the evaluation, we define the following quantities and measures: $t_{\text{stimulus}}$ is the time at the beginning of the invocation of a UPnP action of one of the test devices, and $t_{\text{response}}$ the time a client receives a notification that a state variable changed as result of the invocation of an action. \textit{Latency} is the absolute temporal difference between stimulus and response to an action. This way, latency includes both service control latency (latency related to the invoking of the action) and eventing latency (latency related to the event notification mechanism).

In the context of this test, a device is a simulated OSGi UPnP device, with one service, one state variable \textit{count} of type integer, and one action called \textit{setCount}, which acts as a setter for \textit{count}. Since each device has one service, the terms (UPnP) device and (UPnP) service are used interchangeably. A client is an instance of a WS client whose only function is to record the time, when it is informed about state variable changes. Real clients, such as the visualization client or the composition layer, would of course include more functionalities.

The testing protocol is as follows. After bootstrapping, each client subscribes to \textit{device_count} state variables (of \textit{device_count} different devices). Then, the following step is repeated \textit{iteration_count} times, with increasing values of \textit{current_iteration}: for each device $t_{\text{stimulus}}$ is stored, its action is invoked with \textit{current_iteration} as a value, and then a sleeping time of \textit{sleep_time} ms is issued. Upon receiving a notification that a state variable has changed, each client stores the current time in the $t_{\text{response}}$ which corresponds to the specific state variable and iteration (as reflected by the received value \textit{current_iteration}). Finally, the time measurements are aggregated to compute latency.

### 6.2 Test results

Figure 6.1 shows how the performance of the system behaves when the number of clients increases, by plotting the average latency. As one can see, for up to 30 clients the average latency is well within acceptable bounds (20ms) and then grows very fast.

When increasing the number of clients ($\text{iteration_count} = 10, \text{client_count} = 10, \text{sleep_time} = 100\text{ms}$) the average latency stays is low (less then 6ms) even for 2000 devices.
The evaluation indicates that the tool can support many clients and a high number of devices, still providing very low response times.

### 6.2.1 Analysis

The pervasive layer adds little latency (less than 20ms) even cases which where the number of clients and devices where far beyond what is expected for SM4All scenarios. What becomes quite clear however that increasing the number of clients has a bigger impact on performance then increasing the amount of devices. This is a good thing because the amount of clients is expected to stay reasonably low (less than 10), while the amount of devices may increase at runtime. It is good to note that clients are just one way to extend the system and that the system can also be extended using OSGi bundles, which have much less communication overhead (see section 5.2). Given these facts we can conclude the pervasive layer scales with an increasing amount of devices and performs good with an number of clients and devices that are greater than that of the most complex SM4All scenarios. This means it meets the second criteria stated int the introduction: “Perform well for at scenarios of the scale of the most complex SM4All scenarios”.

It the current time it is not possible to say much about the system performance of the SM4All system as a whole because important issues such as concurrency management and security are not implemented yet, and these things might also add some latency.
Chapter 7

Conclusion

We have designed, implemented and evaluated a Service-Oriented middleware. This middleware was designed with the goals of the SM4All project in mind. To reflect back on the research question.

“Can a Service-Oriented middleware be useful, in the context of the SM4All project, to implement a scalable system that allows interoperability between heterogeneous devices.”

In order to answer this question it helps to assess if the Service-Oriented middleware implementation meets the criteria we set in the introduction. These criteria are:

1. The ability to visualize a simulated scenario, similar to the scenario described in [16].

2. Perform well for at scenarios of the scale of the most complex SM4All scenarios.

The implementation enables the visualization of a simulated scenario, see http://www.youtube.com/watch?v=q4wnm4MaUQ for a video of this simulation. This means the first criteria is satisfied. The second criteria is evaluated in chapter 6. The results presented in this chapter indicate that the pervasive layer is able to scale well beyond what is necessary for even the most demanding SM4All scenarios, and thus meets the second criteria.

Because the system fulfills these criteria we can say with some confidence that this middleware implementation provides both the basic functionality needed and the necessary performance and scalability to be useful in the context of the SM4All project.

To achieve a complete, robust, scalable and user-friendly solution for the SM4All project, many research challenges remain open. Some related to the pervasive layer and some to other components of the SM4All system.

Adding driver for other technologies then UPnP such as Bluetooth and ZigBee as well as providing some way to deal with concurrency are important open issues for the pervasive layer. More test of with different scenarios and more complex hardware devices are also important in order to fix problems and get a better indication of its ability to handle real world scenarios.

Many research challenges remain open for others parts of the SM4All project as well. Improving the planner used for composition, security, privacy, and user interfacing are important topics currently investigated by the partners of the SM4All project. Another direction of future work involves further automating the process of transforming the pervasive-level services to planning-level actions by using an ontology that provides the necessary semantic annotations.
Bibliography

[1] Sm4all project site - http://www.sm4all-project.eu/.


[22] Leonardo Querzoni Roberto Baldoni, Massimo Mecella. D2.1.a the sm4all architecture (version 1). *SM4All deliverables*, 2009.


