
Development of a HTTP-compliant Semantic Architecture for Enhanced Autonomy in Sensor Networks

Master Thesis

Author : Michael Hänni
haennimi@gmail.com

Supervisor : Vlad M. Trifa

Prof. Friedemann Mattern
Institute for Pervasive Computing
Department of Computer Science
ETH Zurich

February 2009

Abstract

At the present time computers minimize their size to dust while keeping sufficient computing power, storage space and battery capacity. At the same time the cost for these little devices shrinks in the same speed as their size. This brings an enormous potential for computer science with revolutionary application scenarios.

On the other hand however this trend raises exceptional problems which come with all the possibilities. The integration of hundreds of embedded devices into one room urges the need of a highly scalable technological infrastructure handling their discovery, their configuration and their maintenance. An important question today in the context of pervasive computing is what technology will provide an integration of such devices. Many researches have been done in the conception of new technologies, which fulfill the demanding requirements for these new applications. A special requirement for such technologies is their user friendliness, especially for non-computer scientists users. Another followed approach is the use (often in combination with some extensions) of existing technologies which have proven their user success.

Our work is a step towards the vision of the "Web of Things" where every day objects equipped with an embedded computer are integrated into the Web. We present our resource-oriented architecture to which embedded devices can be connected. This framework includes a simulator which allows users to simulate pervasive computing scenarios with only very little effort. Our simulator allows users to replace simulated devices with real embedded devices within the same simulation. This results in a seamless transition from a purely simulated to a completely deployed simulation.

We hope that our work will provide a solid basis for other computer scientists to investigate this very promising approach in this scientific field which will change human life in the near future.

Table of contents

1	Introduction	1
1.1	Project Context	1
1.2	Project goals	3
1.3	Contribution of this project	3
1.4	Outline of the report	4
2	Infrastructure for the Web of Things	5
2.1	Web technologies	5
2.1.1	WS	6
2.1.2	REST	6
2.1.3	Comparison and Conclusion	7
2.2	Simulation Technology	8
2.2.1	Network and Device Simulators	8
2.2.2	Multi-agent Simulation	8
2.3	Ontologies	9
2.3.1	Ontology Technologies	9
2.3.2	Device Representation Ontology	10
3	Related Work	13
3.1	Web Technologies for Embedded Device Architectures	13
3.1.1	Architectures for Heterogenous Devices	13
3.1.2	Integration of Physical Objects into the Web	14
3.1.3	Web enabled Devices by Design	14
3.2	Simulation Frameworks	15
3.2.1	Discussion	16
3.3	Ontologies for the Semantic Web	17
3.3.1	Discussion	18
4	Smart Gateways	21
4.1	Addressing	21
4.2	Gateway registration	22
4.2.1	Heartbeat	22
4.2.2	Link recovery	22
4.3	Device Listing and Description	23
4.4	Eventing	23
4.4.1	Keywords for Semantic information	23

4.4.2	Client registration and event refresh	24
4.4.3	Event installation	25
4.4.4	Event firing	25
5	Smart Simulator	27
5.1	JADE	27
5.1.1	Agent Platform	27
5.1.2	Agent	28
5.1.3	Behavior	28
5.1.4	Communication	29
5.2	Simulator components	29
5.2.1	Simulation Controller	29
5.2.2	Device Agents	30
5.2.3	Internal Agents	32
5.2.4	Simulation Behavior	32
5.3	Smart Simulator Application Scenarios	32
5.3.1	Pure Simulation	33
5.3.2	Hybrid Simulation	34
5.3.3	Deployed Simulation	34
6	External Client	37
6.1	Browser	37
6.2	Administration Tool	37
6.3	External Client Application	38
6.3.1	Architecture	39
7	Sample Application	41
7.1	Hotel Scenario	41
7.1.1	Smart Gateway Structure	42
7.1.2	Devices	42
7.1.3	External Client	43
7.2	Create your own Application	43
7.2.1	Smart Simulator	45
7.2.2	Smart Gateway	46
7.2.3	External Client	46
8	Conclusions and Future Work	47
8.1	Future Work	47
8.1.1	Security	47
8.1.2	Use case studies	48
8.1.3	And now for something completely different...	48
8.1.4	Ontologies	49
8.2	Conclusions	50
Appendix A : Sample Simulation Device Agent		51
Appendix B : Sample Device Behavior		55
Appendix C : Sample Simulation Device		57

Appendix D : Sample External Client Panel	59
Appendix E : Sample Smart Gateway Driver	63
Bibliography	71

Introduction

"I have always wished for my computer to be as easy to use as my telephone. My wish has come true because I can no longer figure out how to use my telephone."

Bjarne Stroustrup (1950-)

This chapter introduces the context of our work and lists the major parts of it.

1.1 Project Context

Sensor nodes, or *motes*, are often used as embedded devices for implementing showcases applications in the field of pervasive and ubiquitous computing. Such devices include BTnodes¹, Java SunSPOTS², or Mica³ nodes and do often have similar capabilities: They are equipped with temperature, light, acceleration, or noise sensors and have an integrated radio module for communication. For computing tasks they have a micro controller with on-chip and external memory. Based on their sensing capabilities showcase applications taking such sensor nodes as embedded devices are therefore naturally predominantly scientific ones. The inspiring vision formulated by Mark Weiser (Weiser, 1991) aims at everyday objects which are supposed augment our daily personal and business lives. It becomes immediately clear, that an intelligent plant pot would have to be outfitted with other sensors as mentioned above in order to become useful. The most important sensors for this plant pot would be a soil humidity sensor to know whether the plant needs more water and further a light sensors for knowing whether the plant has too much, enough, or too little light. The required sensors for a room lamp would be completely different. An intelligent lamp wouldn't really need any sensors by itself, but rather need some actuators for reacting in an intelligent way to environment changes. The integration of

¹See <http://www.btnode.ethz.ch/> .

²See <http://www.sunspotworld.com/> .

³See <http://www.xbow.com> .

such augmented objects into our lives requires a capable technological infrastructure. Thinking of the huge amount and divergence of every day objects leads to a single generic underlying technology where devices with all sort of capabilities connects to and communicates with. The idea behind this credo is to use one powerful technology for many heterogenous devices.

The reason why the existing sensor nodes have the mentioned generic capabilities to be open for as much applications as possible. Therewith producers reach high production numbers which lowers the production costs per device. On the other hand the lack of a accepted and trustworthy standard sets a high barrier for producers of everyday goods for supplying their products with computational capabilities. So there is a clear need for a convincing standard which will supply a solid base for any embedded computer making part of any every day object. Here applies our idea to build a simulator which allows rapid prototyping of pervasive and ubiquitous scenarios. By defining the capability and placement of the devices one can simulate hypothetical devices having any capabilities. Of course, this demonstrator would have to be integrated into what we see as a promising base technology for the integration of any embedded device.

The base technology for the integration of embedded devices is an ongoing discourse. As we will see in chapter 3 many different approaches have been presented for new architectures that aim to be adequate for integrating embedded computers. But most projects are too inflexible and specific for being suitable for a generic set of embedded devices. There is little enthusiasm for agreeing on a completely new architecture which is powerful enough in terms of generality and scalability. Currently there does not seem to raise a new technology where manufacturers and developers will agree on.

Maybe a completely different approach will lead us towards the vision of a "Web of Things". Why not use a highly successful technology used by approx. 1.5 billion⁴ users and connects an even considerably higher number of devices. The Web is a tremendous success story. Starting in 1969 serving only a dozens of computers, the Internet technology has proven its genericity and scalability in an impressing manner. Not only the rapid growth, but also the integration of new technologies and devices, which have shown also the flexibility of this technology suggests to be a good basis for the integration of embedded devices.

In our work we have chosen to explore the second alternative and made a conception of an underlying architecture using the Web standards and integrate the simulator into this architecture. The conception of the architecture was done by Samuel Wieland and Michael Hänni, the implementation of the architecture was done by Samuel Wieland and the implementation of the simulator was done by Michael Hänni. The work was done as their master thesis and was supervised by Vlad Trifa.

Our project aims to be a contribution towards the vision of a "Web of Things". We introduce our resource-oriented architecture which aims to integrate embedded devices into the Web. In our approach the integration of the devices follows the same principles which were used when "old fashioned" computers were connected. This architectural style integrates resources seamlessly into the internet with an uniform interface. Our architecture strictly follows this principle and thereby lowers the adoption costs for later users. Our work includes a simulator which provides a rapid prototyping platform for pervasive and ubiquitous scenarios. Our simulator allows different types of simulations: Any simulation can be performed from a purely virtual over a hybrid to a simulation where all devices are real hardware devices. The implementation environment remains the same for all the different types of simulation. We have defined a device representing ontology for their better integration. This ontology defines at the present time the structure in which devices are represented. In a future work this formally defined representation can be used and extended for automatic context sharing and reasoning

⁴See <http://www.internetworldstats.com/stats.htm> .

among devices. For defining a sample application of our framework, we have implemented a scenario located in the hotel business. Our scenario simulates or deploys embedded devices in a sample hotel. These devices provide for instance guest rooms with the ability to determine the location of their guests or equip building units with fire control informations. As a result of the augmented rooms and places in the hotel many different applications can use the generated informations for their purposes.

1.2 Project goals

The aim of this project is the conception and the implementation of a simulator which allows the rapid deployment of an application scenario with the minimum of configuration and implementation effort for a developer. The simulator has to be integrated into an architecture which provides a qualified basis for the integration of embedded devices into the Web. Here are the most important parts which have to be designed and implemented:

HTTP-based Interaction Propose a simple HTTP-based system for semantic annotation and reasoning for physical devices to specify their interactions and functionality in a dynamic context (unknown devices with different capabilities continuously appear and disappear).

Multi Agent Simulator Develop a simulator based on a multi-agent system. An essential requirement will be that simulated devices will communicate together using standard Web technologies (over simulated HTTP) so that code can be easily ported on any device that can be accessed over HTTP. Implement a prototyping application with several devices that run the proposed semantic model and illustrate the advantages offered by the method. If time allows, perform a user study to test the system with human users.

Ontologies Investigate and evaluate different semantic standards (microformats, OWL, SWS, RDFa, ontologies) and how they can be used for physical computing devices. Implement and simulate a large-scale scenario.

1.3 Contribution of this project

Besides the conception as a theoretical basis, the implementation of the Smart Gateway architecture with the integrated Smart Simulator offers a generic framework for both the simulation and the deployment of pervasive computing scenarios towards the "Web of Things". In the same environment scenarios can in a first step be simulated, in a second step actual physical devices can be integrated into a hybrid simulation and finally all simulated devices can be replaced with physical devices. In this transformation from a pure simulation to a real deployment of a pervasive computing scenario there is no need to change the environment. We hope that the experience of such sample applications will show the genericity and scalability of our architecture. The resource-oriented approach building upon well-known Web principles guarantees a very low adoption barrier for potential users and an excellent reusability.

Web application developer With the use of our framework embedded devices can be integrated into the Web. Web application developer can compose completely new application scenarios without having to concern about the integration of the devices. As part of a case

study we could integrate a set of embedded devices as resources into the Web and invite Web developers to write some applications for them. The resources could be simulated or real ones.

Web end user The Smart Gateway architecture demonstrates a possible way to integrate embedded devices into our daily life by integrating them into the Web. Regular Web users could benefit from the newly integrated resources and consume the newly generated applications.

Producers of embedded devices Producers of embedded devices receive a simulator which allows them a rapid prototyping of scenarios where their goods would be virtually equipped with specific capabilities. In a simulation producers can estimate the potential benefits and drawbacks of their augmented goods. Devices for building monitoring for example are very much suited for our approach as the hierarchical building structure can be mapped to a Smart Gateway structure. In this structure then devices can be positioned and scenarios can be simulated. At first purely virtual and later in combination with real devices.

1.4 Outline of the report

This report is organized as follows:

Chapter 2 In this chapter technologies which were used as a basis for our work are introduced. The chapter also contains motivations for decisions on what technologies to build on.

Chapter 3 This chapter analyzes some related works, and puts them in relation with our work. Related works have been analyzed at different layers.

Chapter 4 The conception of smart gateways. This part describes the architecture behind the Smart Gateway architecture as well as the features the structure offers.

Chapter 5 The Smart Simulator is introduced. Using the Smart Simulator not only simulated devices and be created, also a Smart Gateway structure can be instantiated. The chapter describes the architecture behind the simulator and its features.

Chapter 6 The External Client chapter introduces different types of clients of our framework.

Chapter 7 The sample application. The sample application brings the Smart Gateway, the Smart Simulator and the external clients to work and demonstrates a sample application for the designed framework. As second part of this chapter there will be a description of how to implement a custom scenario using the Smart Simulator.

Chapter 8 Conclusion and future work details the different aspects of our work to get an idea of benefits and drawbacks of our implementation. Future work mentions further steps which seem to be promising or further investigations.

Infrastructure for the Web of Things

"When you don't invest in infrastructure, you are going to pay sooner or later."

Michael Parker (1949-)

This chapter gives a brief overview of the main concepts of the implemented work. For every component of our work the chosen underlying technology is presented along with some alternatives.

2.1 Web technologies

Our decision to use Web technologies as a base for our framework was an early and major decision we had to take. As alternative we have looked for an architecture which is designed specifically for pervasive computing. Web technologies have however proven its flexibility as a successful base technology in the applications of the so called Web 2.0, which go in completely different directions as the primary client-server based scenarios for which the Internet technologies were designed for. So we decided to use Web technologies as base of our framework and to pursuit the "Web of Things" vision.

After that we had to think of an appropriate architectural style from the Web technologies. The famous "SOAP vs REST" or "WS-* vs REST" debate is ongoing for several years and has taken almost religious traits. Only few works compare the two directions in an unagitated, scientific manner. As for all such debates neither the first nor the last architectural style outperforms the other in every application scenario. So it is more important to identify their strengths and weaknesses in order to judge which style addresses the specific needs for a given application. A detailed analyzation and comparison of the two styles has been done in ([Pautasso et al., 2008](#)). Here we try to identify the strengths and weaknesses of each approach. Followed by this we map these characteristics to the context of our project and motivate our decision for REST as an architectural style.

2.1.1 WS

WS-* uses HTTP as transport layer protocol for Remote Procedure Calls (RPCs). A service is provided through a network addressable software component. All WS-* services have the same technical base:

SOAP The Simple Access Object Protocol is a XML based language which defines the message architecture and formats. XML Schema describes the structure of the message, so that the SOAP engine at the two endpoints is able to un-marshall and marshall the message content and deliver it to the proper implementation. The SOAP document consists of an envelope, which contains a header and a body. The header is extensible and can be used for routing or quality of service configurations. The body carries the payload of the message.

WSDL The Web Services Description Language defines the communication interface between the service provider and the service customer. A WSDL has a *port type*, which contains the operations which are associated with incoming or outgoing messages. The *binding* defines a transport protocol to the operations.

Strengths WS-* based services have brought interoperability between heterogeneous middleware systems. A benefit which results in using WS-* is the protocol transparency and independence. Having this property features like encryption or reliable data transfer can be made "end-to-end" and independent of the transport layer. WSDL supports synch- and asynchronous interactions. Further the distribution of the service partners is abstracted as the remote service seems to be at hand with normal procedure calls. A enormous variety of tools hide the complexity of WS-* services from the user.

Weaknesses In the last mentioned benefit lies also a potential danger: A tool hiding several aspects of the architecture can lead for a developer to loose the overview. Typically tool generated services also have much lower performance than "hand made" services. This is weighing in heavier in our context as embedded devices have significantly limited computing resources. Further interoperability problems can rise when native data types of a language are part of the service interface as the different endpoints may have been built using different programming languages. The translation between XML and a programming language, mainly when complex data structures are exchanged, are very complex such this translation stays a constant error source with the often need of cumbersome solving effort. Finally, WS-* services are often very heavyweight, which is a huge problem in the context of pervasive computing.

2.1.2 REST

Representational State Transfer, or *REST*, is an architectural style which was introduced in (Fielding, 2000). HTTP is an application layer implementation of this style. REST aims to return to the root principles of the Web technologies. The central concept of the architecture are resources. A resource is any source of information. Actors of REST applications take roles following the client-server principle and interact with resources. We present the principles of rest and show how they are implemented in HTTP:

Resource identification All resources have a unique identification. In the HTTP implementation this is done using URIs (Berners-Lee *et al.*, 2005).

Uniform Interface The manipulation of resources is done using a fixed set of operations. In HTTP there are four operations which are characterized by the verbs create, read, update and delete and called PUT, GET, POST and DELETE.

Self descriptive messages The representation of a resource is separated from its content. In this way the content of a resource can be accessed in a variety of formats. HTTP allows formats such as HTML, XML, plain text, PNG, etc.

Stateless In REST all interactions with resources are stateless. In the HTTP implementation this principle can be violated through techniques as cookies.

Strengths A clear strength of RESTful Web-services lies in its simplicity and popularity. The simplicity comes from the uniform interface which does not change, and the popularity of the HTTP implementation. As the Web standards are very common, there is a very little barrier for adoption. The World Wide Web, which is built following the REST principles, proves the extraordinary scalability of this approach and the ability to discover resources without a centralized repository. RESTful Web services are extremely light-weighted. This is an ideal characteristic for embedded devices with very limited computing power. Services can be implemented with a minimum of tool support, and no test-client has to be implemented, as a simple Web server can be used for this task.

Weaknesses When a complex interactions with resources are needed, then the REST approach can become cumbersome as the only way to interact with resources are HTTP calls. The uniform interface is also an absence of a clear contract between client and server. The server may change the representation of a resource without telling the client. Finally the absence of a commonly accepted marshaling mechanism for URIs leads to a great challenge when complex data structures need to be encoded.

2.1.3 Comparison and Conclusion

A further interesting work to read concerning the comparison of REST and WS-* is ([Vinoski, 2008](#)). The author claims, that REST services are much more likely to being reused. He points out, that the more specific a service interface is, the less likely it is that it will be reused. WS-* services tend to be highly complex and specialized. If now requirements change, then costly efforts need to be made in order to adjust the service to the new situation. The author claims that if specialized interfaces hinder reuse, then by minimizing the differentiation to a universal interface, such as REST offers, the reuse will be increased. Further simplicity and extensibility increase too. But the most important argument in the context of our scenario is that an uniform interface plugs your resources into the Web framework in a consistent and instantly usable way for a wide variety of clients. "Reuse the Web and you thereby increase the chances that what you put there will be reused." ([Vinoski, 2008](#))

Following these arguments we decided to chose REST as the architectural style for our framework. To use the resource-oriented approach of the Web is more obvious than the service oriented approach of WS-* based architectures. We claim that the integration of embedded devices should be done using the same approach which was used for the integration of web servers into the Web. The uniform interface provides a very simple way for clients to interact

with the resources. We believe that this decision to reuse and extend the Web will bring a promising way to integrate smart, embedded devices into our daily lives.

For the implementation of our framework we used RESTlet¹ as a lightweight REST framework.

2.2 Simulation Technology

To simulate the embedded devices for the pervasive application scenarios we needed a base technology, which allowed us to simulate a large quantity of devices. Our focus in our search was a framework which allowed to simulate autonomous devices with a specified behavior without the need of a cumbersome learning period before using it. We saw multi-agent simulation as very suitable for this task but also examined other simulation tools.

2.2.1 Network and Device Simulators

As a alternative to a multi-agent simulation, we thought of using a network or device simulator instead. We give here a short description of device and network simulators and motivate why a multi-agent simulator is more suitable for our task.

Network Simulator A network simulator such as ns² simulates network entities as computers, routers, hubs, mobile units, etc. The focus of such simulators is however network traffic or load balances. A further usage is the testing of new protocols.

However in our project we focused on the *behavior* of autonomous devices, not on the messages sent in various directions. The focus lies on the different devices with their interpretation of the world and their interaction with the environment.

Device Simulator Many embedded devices such as Java Sun SPOTs³ come with a simulator where program code written for such devices can be simulated on one or more virtual devices. The drawback of such simulators is the restriction to the concrete embedded device for which instances can be simulated. For our project this would have meant a too big restriction.

2.2.2 Multi-agent Simulation

Multi-agent simulation tools are used for simulating distributed applications. Apart from computer science where many applications are concerned with artificial intelligence, the systems are also used in biology and social sciences. The basic concept of these systems is an agent, which is a piece of software and has these typical characteristics:

Autonomy Each agent acts autonomously. Its behavior is proactive working on a certain task or by sending messages to other agents or reactive by the receipt of a message from an other agent. The workflow of an agent can not be controlled otherwise by sending messages to it.

Local knowledge An agent has only local knowledge there is no central entity with global knowledge. For sharing knowledge agents can send messages to each other.

¹See <http://www.restlet.org> .

²See <http://www.nsnam.org> .

³See <http://www.sunspotworld.com/> .

Decentralization As all agents are autonomous and have only local knowledge, the system has not a central entity which collects informations or operates the agents.

As our focus is to simulate embedded devices with sensing and actuating capabilities which follow a specific behavior specified by a piece of software, the use of a multi-agent simulation tool as base for our simulator was an easy and early decision.

We had a look at different frameworks and interviewed some people with experience in such frameworks. The following chapter details these activities.

2.3 Ontologies

A detailed survey for getting an overview of the benefits and concepts of ontologies in computer science is (Ye *et al.*, 2007). The authors define the usage of ontologies ”to build consensual terminologies for the domain knowledge in a formal way so that they can be more easily shared and reused”. Some key aspects of an ontology are:

Explicitly The concepts and constraints are explicitly defined.

Formalization Each concept is formally defined in terms of its meaning. Axioms constrain the interpretation and well-formed use of the concepts.

Sharing Ontologies capture consensual knowledge.

The authors further classify ontologies in two dimensions. The first classification is done using its generality:

- **generic ontologies** describe general concepts, individual of a specific domain.
- **domain ontologies** describe concepts for a specific domain (such as, Biology).
- **application ontologies** describe concepts for a specific application. Application ontologies can build upon generic and domain ontologies.

The second classification considers the level of expressiveness:

- **lightweight ontologies** include concepts, taxonomies and relation between the concepts.
- **heavy ontologies** apply axioms and constraints to lightweight ontologies. This allows semantic interpretation for the concepts.

These two classifications help to compare and distinguish different ontologies and their working point.

2.3.1 Ontology Technologies

Here the most popular ontology technologies are listed. They all are W3C recommendations for the encoding of ontologies.

XML The eXtensible Markup Language is a standard for describing data in a semi-structured manner. XML Schema or Document Type Definition (DTD) were introduced to constrain the syntactic structure of an XML document.

RDF The Resource Description Framework (RDF) is built on top of XML. It defines structural constraints in order to express semantics. Statements in RDF are unambiguous and consist of a subject, a predicate and an object. The subject is a resource represented by an URI. The object is another resource and the predicate defines a relation between the subject and the object.

DAML+OIL DAML+OIL is a composition of the Ontology interchange Language (OIL) and the DARPA Agent Markup Language (DAML). DAML+OIL supports data structures, well-structured semantics and inference procedures based on description logic.

OWL The Web Ontology Language is basically an extension of DAML+OIL with additional vocabulary and formal semantics. There are different flavors of OWL: OWL lite, OWL DL and OWL full. The different versions have increasing expressiveness. The most commonly used is OWL DL.

2.3.2 Device Representation Ontology

For our framework we have we have designed a device representation lightweight application ontology based on the REST architectural style. We decided not to use ontologies for augmenting the intelligence of the device, but, as a first step to define an ontology which declares an interface for the addressing of the different resources. Our ontology is motivated by the Delivery Context Ontology⁴, a W3C working draft. We decided to create a similar but much thinner ontology. The implementation of the whole standard would not have included our specific requirements, but would have been far too detailed for our needs.

The figure Fig. 2.1 shows the structure of our device representation. If for example the host `myHost.com` has attached a device `myDevice` a client can access the representation of the device `myDevice` with a simple web-browser by entering `http://myHost.com/myDevice`. Further examples of a browser based access of devices in our ontology are listed in chapter 6.

⁴See <http://www.w3.org/TR/dcontology> .

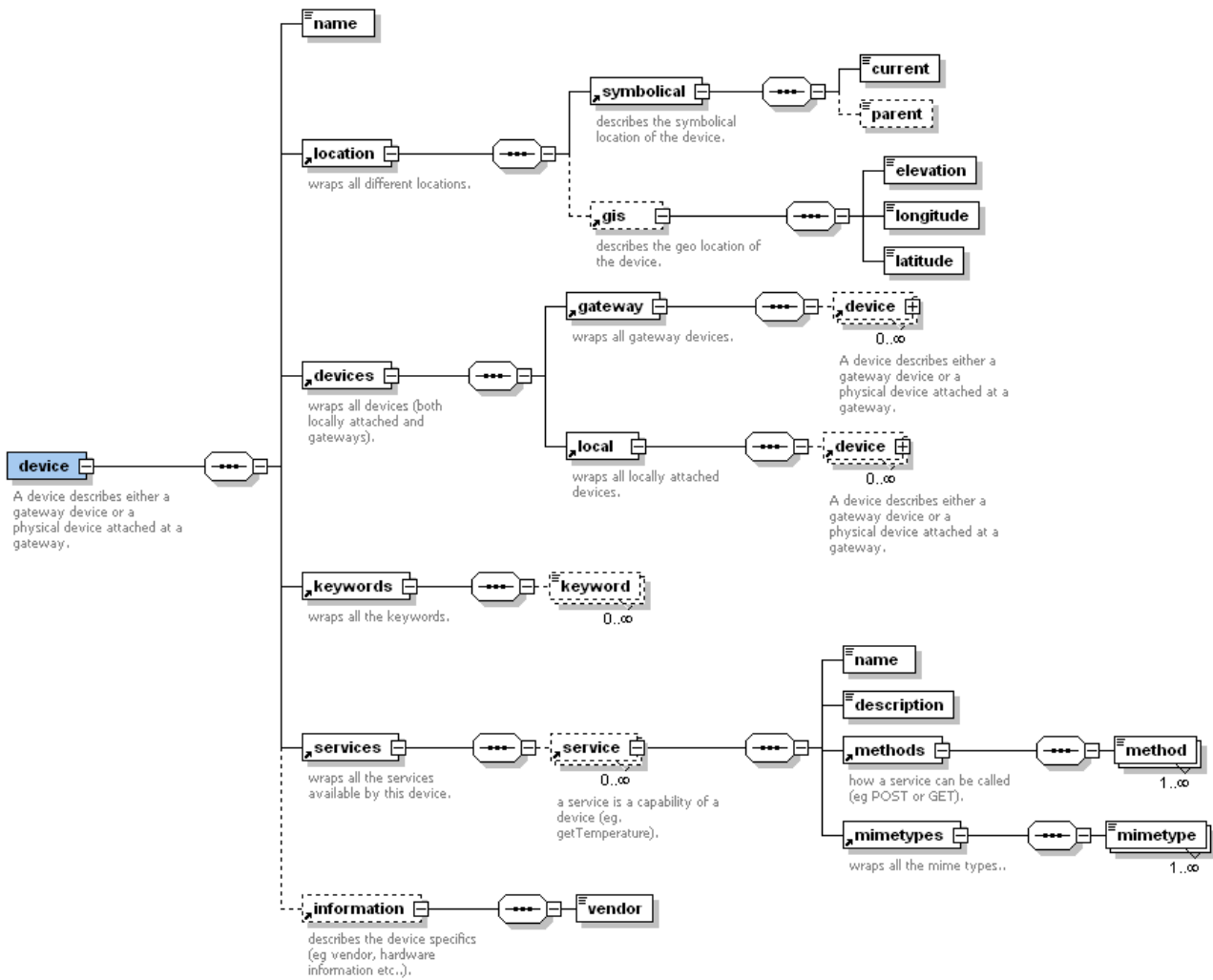


Figure 2.1: The REST based representation scheme of a device in our framework.

Related Work

"The only place where success comes before work is in the dictionary."

Donald M. Kendall (1921-)

In the following some related works shall be introduced. First some works in the area using Web technologies for embedded devices are presented, then some multi-agent simulation frameworks are introduced. Later we have a look at some ontologies for the semantic Web.

3.1 Web Technologies for Embedded Device Architectures

The vision of the "Web of Things" which aims to integrate every day objects into the Web. In the following we highlight some steps which have been done in order to reach this vision and compare them with our approach.

3.1.1 Architectures for Heterogenous Devices

To find a accepted standard for the cooperation of heterogenous devices many middleware architectures such as CORBA, DCOM or RMI have been proposed. In the context of embedded devices Wang et al. make a survey ([Wang et al., 2008](#)) between different proposed middleware for wireless sensor networks (WSN). In recent years Service oriented Architectures (SOA) mostly in the context of web services have become very popular. Recent works in the field of pervasive computing have proposed SOA based architectures in order to integrate embedded devices into the Internet. Architectures such as IrisNet ([Gibbons et al., 2003](#)), or GSN ([Aberer et al., 2007](#)) provide a platform for the configuration, collection and maintenance of sensors networks. Such architectures succeed in combining different types of devices into one application and bring the pervasive computing scenarios to a different level of abstraction.

The major drawback of these architectures is however their inflexibility. Based on the concept of the Remote Procedure Call (RPC) the coupling between the devices and the middleware is API - based. This results in a tight coupling which brings unsolvable problems for flexible and mobile interacting scenarios. The building of a network cannot be done using an inflexible base. Such a network will have to be replaced by another one in case of a change of requirements or usage scenarios. Therefore a much more promising approach lies in the integration of embedded devices into the Web.

3.1.2 Integration of Physical Objects into the Web

The integration of embedded devices into the Internet has been done using different approaches. Starting with the creation of a virtual counterpart in works as Webstickers (Ljungstrand *et al.*, 2000) or Cooltown (Kindberg *et al.*, 2002). The creation of virtual counterparts of physical objects was restricted to a description of the item. There was no interaction of usage of the functionalities of the object possible.

A next step have been services, where sensors can be shared. An infrastructure for sharing sensors which are distributed all over the world is SenseWeb (Kansal *et al.*, 2007). Typical applications of SenseWeb are data collecting scenarios. The collected data items are then interpreted and processed by a central entity of the application. The sensors making part of the infrastructure are however not really integrated into the Web. They merely send data to a central server using a special software which has to be installed for using the infrastructure.

An other approach is Pachube¹, where people share their sensor data with the community. In particular Pachube offers the service that sensor data are available at realtime and can be obtained using RSS feeds. Apart from realtime informations, the whole history of sensor values can be tracked using a central repository. As in SenseWeb, we see here a major drawback of Pachube for the integration of embedded devices into the Internet. A centralized architecture will not be able to scale a vision of Web of Things.

3.1.3 Web enabled Devices by Design

In our approach we intend to use the Web as application layer. Our work seamlessly integrates the embedded devices into the Internet by following the same architectural principles on which the Web itself is built upon. These REST principles define a loose coupled system with an uniform communication interface. Wilde proposes such an approach in one of his articles (Wilde, 2007). He states, that by following this way the integration of physical resources will be as seamlessly as possible. He highlights one of the most important strengths of the current Web in the possible recombination of the same resources for different applications. Also the currently very popular integration of new resources to an existing one could become a very interesting "physical mashup" in the context of physical devices with completely new application scenarios.

Following the REST principles the integration of embedded devices will bring a capable base technology for the vision of the "Web of Things". The concept of uniform interfaces will provide a very low adoption hurdle for future users to reuse such an infrastructure. Because of the extremely scalable architecture different applications will be able to interact concurrently with overlapping sets of resources. For future visions of intelligent entities the RESTful integrated devices will provide a solid basis for the appliance of semantic technologies on top of it.

¹See <http://www.pachube.com> .

3.2 Simulation Frameworks

The Multi-agent simulation in computer science applies the concept of multi-agent systems in simulation. Active components of the monitored system are viewed as agents. Their behavior is individually specified. Thereby it is possible to simulate certain phenomena and dynamic interactions between agents.

There are many tools and frameworks for modeling and simulation of agent systems. Some important at which we had a look are:

- **Swarm** Originally developed at the Santa Fe Institute². Now being developed at a private, non-profit organization.
- **SeSAM** Developed at the University of Würzburg in Germany.
- **AnyLogic** Distributed by the XJ technologies company.
- **Jade** Developed by Telecom Italia.
- **MASON** Developed at the George Mason University.
- **NetLogo** Authored by Uri Wilensky a professor at the Northwestern University³.

We tested these frameworks against a set of criteria:

- **Open Source** The framework should be open source, as we do intend to make our framework also open source.
- **Java Compatible** As the Smart Gateway structure is implemented in Java and we want the framework to run on any platform we decided to use Java.
- **Good Documentation** To minimize the learning effort we want the project to supply a good documentation.
- **Stable Version** We do not want to worry about bugs in the framework.
- **State-of-the-art Framework** We want the framework to be scalable, robust and up-to-date with modern technology.

Swarm Swarm⁴ is an open source platform where applications can be written using Objective-C or Java. The software packages can be deployed on Windows, Unix or Macintosh. The last stable release (version 2.2) dates back to February 2005. Swarm applications can be implemented inside the Eclipse IDE. The swarm architecture has no specific application focus but is for general purpose agent based simulation.

SeSAM SeSAM⁵ stands for Shell for Simulated Agent Systems and was developed at the University of Würzburg. Being in constant use at the university the framework has a quite active development and community although mainly restricted to the university. The latest stable release was version 2.5 at January 2009. SeSAM is open source and written in Java. The main purpose of the project is the teaching of agent based models for students at university.

²See <http://www.santafe.edu/> .

³See <http://www.northwestern.edu> .

⁴See <http://www.swarm.org> .

⁵See <http://www.simsesam.de> .

Repast Repast⁶ (Recursive Porous Agent Simulation Toolkit) is an open source toolkit which is specialized for applications in the context of social sciences. The project is implemented in Java, the latest stable release was on December 2008. Repast runs on Windows and Mac and has an Eclipse plugin. The project is developed by a team of docents developers. There is however no forum but a (mainly filled with spam) mailinglist.

AnyLogic AnyLogic⁷ is a proprietary product written in Java. A educational faculty license costs 575 Euros. As this product os not open source, we decided against further investigations.

JADE JADE⁸ (Java Agent DEvelopment Framework) is an open source framework written in Java. The latest version of JADE (3.6.1) released on November 2008. The framework has a extensive list of documentation and tutorials. There is also a very active mailing list for developers. JADE targets t oa minimal footprint and can also run on mobile embedded devices which have mobile Java environments.

MASON MASON⁹ Stands for Multi-Agent Simulator Of Neighborhoods. It is implemented in Java and open source. MASON is a joint effort between George Mason University's Evolutionary Computation Laboratory¹⁰ and the GMU Center for Social Complexity¹¹. The direction of the simulator is to visualize problems life the game of life.

NetLogo NetLogo¹² is a free, but not really open source platform for multi-agent modeling. Some copyright notices have to be included for redistribution. Latest release (version 4.0.4) was on November 2008. The author is Uri Wilensky a professor at the Northwestern University. The framework is written in Java and applications runs as standalone applications or as Java applets inside a Web browser. The platform goes mainly in the direction of modeling social and natural phonemes.

3.2.1 Discussion

After collection some basic informations about the mentioned tools, we sorted out two kinds of frameworks: Firstly, the commercial products as we intend to redistribute our framework, secondly, we decided to sort out all tools which have a focus on visualizing social and biological phonemes as we intend to simulate devices not on the graphical output of a simulation. The collected basic informations about the frameworks are listed in the table 3.1.

In a second step we focused on the remaining tools: Repast and JADE. We implemented a small example application with both tools. Our experience was, that in the JADE framework the concepts are very intuitive and easy to learn. In the sample Repast application, we spent more time in getting an overview of the different components. For this reason we decided to use JADE. The framework is introduced in detail in chapter 5.

⁶See <http://repast.sourceforge.net> .

⁷See <http://www.xjtek.com> .

⁸See <http://jade.tilab.com> .

⁹See <http://cs.gmu.edu/~eclab/projects/mason> .

¹⁰See <http://cs.gmu.edu/~eclab> .

¹¹See <http://socialcomplexity.gmu.edu> .

¹²See <http://ccl.northwestern.edu/netlogo> .

framework	pros	cons	score
Swarm	Eclipse plugin	last release 3 years ago	-
SeSAM	good docu	merely a learning tool	-
Repast	good docu and tutorials	"closed" community	+
AnyLogic		commercial product	- -
Jade	active community; excellent docu		+ +
MASON		visualizing social phenomena	-
NetLogo		copyright limitations	-

Table 3.1: The discussion of the different framework. Key: - - : not suitable; -: limited suitability; +: rather suitable; + +: suitable. Where the suitability is judged regarding the context of our project.

3.3 Ontologies for the Semantic Web

We have seen in chapter 2, there exist different types of ontologies. Also different technologies are used for their encoding. In the following the most popular ontologies are introduced.

A very interesting analyzation and comparison of the most popular ontologies can be found in (Ye *et al.*, 2007). For analyzing ontologies the authors propose a list of criteria to assess ontologies:

- **Clarity** Terms must be unambiguously and communicated effectively.
- **Coherence** Definitions must be consistent.
- **Ontological commitment** Make just enough claims to support the intended knowledge sharing and reuse.
- **Encoding bias** Ontologies should not depend on a particular symbol-level encoding.
- **Extensibility** It should be easy to add new terms without affecting existing ones.
- **Orthogonality** General concepts should be defined as independent and loosely coupled atomic concepts.

Further the authors list key elements and compare their representation in the different architectures and for suggest some characteristics for some key elements which a general ontology should support or have in mind. The key elements are:

Location For the authors the most important form of context. Suggestion: The support for different types of representations for locations. Coordinate locations, regions which can be described geometrically, and symbolic locations that encompasses an abstract idea where an object is. The ability to map between physical and symbolic locations The ability to exploit rich spacial relationships like hierarchical spacial relationships or adjacency spacial relationships which define overlapping, adjacency, or disjointedness relationships. Additionally the concept of distance should be introduced.

Agent/Person The agent/person describe the actors of a system. Suggestion: The person ontology is relatively application specific, but should be built with the orthogonality and ontological commitment criteria in mind.

Time Temporal representation and temporal sequence relationships is an important concept, because it is closely related to most contexts. Suggestion: A good temporal ontology should consider the relationship between existing between physical time and support the mapping between symbolic and physical time.

Activity Any action that an actor of a system can perform should be modeled. Suggestion: Four orthogonal types of activities which all should be supported. Each activity can be defined from different perspectives: The source of provided activity information. The function of the activity (sensing, providing a service, or scheduling). The duration of the activity. The location of the activity.

In the following five popular heavyweight domain ontologies for pervasive computing are introduced:

CoBrA ([Chen et al., 2003](#)) The Context Broker Architecture (CoBra) provides a means of acquiring, maintaining, and reasoning about context, sharing knowledge and detecting and resolving inconsistent knowledge. The central point of intelligence in CoBrA is the context broker. There exist different sets of ontologies which are implemented in OWL. SOUPA is an ontology set which aims to support pervasive computing applications.

Gaia ([Román et al., 2002](#)) The main characteristic of Gaia is that it brings the functionality of an operating system to physical space. Physical and virtual entities are able to interact seamlessly. The ontologies make Gaia systems context aware and are maintained by the Ontology Server. Ontologies are written in DAML+OIL documents. The Ontology Explorer is a GUI that allows human users to get a better understanding of the semantics in the system.

GLOSS ([Coutaz et al., 2003](#)) A GLObal Smart Space (GLOSS) provides a large and divers range of location-aware services using location informations based on the GLOSS ontologies.

ASC ([Thomas et al., 2003](#)) Aspect-Scale-Context (ASC) is a model for describing contexts and their relationships using ontologies. A context is a set of contextual information characterizing entities. An aspect is a symbol-value whose subsets are a super set of all reachable states. Scales are a representation format of an aspect. Ontologies implemented in ASC facilitate service discovery and service interoperability on the context level.

SOCAM ([Gu et al., 2005](#)) The Service-Oriented Context-Aware Middleware (SOCAM) enables the rapid prototyping of context-aware services in pervasive computing environments.

3.3.1 Discussion

Initially one aim of the Smart Simulator was to include a popular ontology into our framework in order to enable formalized context informations in the devices. Devices integrated into the Web would become context-aware and thereby become intelligent. This intelligence would allow

devices to gain autonomy for solving common tasks or reacting to context changes. However during the implementation we have seen, that the mentioned ontologies are not suitable for the integration into our framework. Ideally an ontology should be a modular plugin, which could be added to devices or gateways of our framework. The introduced ontologies however are too cumbersome for an integration into an existing framework and would have reduced its genericity dramatically.

A different approach to the usage of these ontologies which are written using an ontology language such as OWL, is to add meta information about a Web enabled resource directly into its representation. Such ontologies technologies for the semantic Web include RDFa¹³ (Resource Description Framework - in - attributes) which uses metadata from XHTML to provide a set of attributes. A similar approach is microformat¹⁴ which also puts semantic informations into XHTML or HTML pages using markup tags. In our view these approaches are much more suited for augmenting resources with semantic informations. Also the integration of such techniques into our framework is much more feasible. Prior to the integration of such techniques, the representation of the resources has to be defined. Our device representation ontology is described in 2.3.2. Building upon this ontology, semantic informations could be included into the device representation with the use of techniques such as RDFa or microformat.

¹³See <http://www.w3.org/TR/xhtml-rdfa-primer/> .

¹⁴See <http://microformats.org/> .

Smart Gateways

*"It's not that I'm so smart
it's just that I stay with problems longer ."*

Albert Einstein (1879 - 1955)

This chapter shall give a short overview to the smart gateway presentation layer. Addressing of hierarchical gateways and devices are covered as well as gateway and events registration. The conception of the Smart Gateway architecture was done by Samuel Wieland and Michael Hänni. Samuel Wieland implemented the architecture as part of his master thesis.

4.1 Addressing

Each gateway within the hierarchy of gateways can be addressed in two ways:

1. **direct addressing:** if you know the ip address of the gateway you can interact directly with the gateway. For example if you know the ip address of gateway C in (Fig. 4.1) you can directly call it: `http://GW_IP/`
2. **hierarchical tree addressing:** If you know the location of the gateway in the tree but you dont know the address you can call the gateway through the tree structure (You always know the ip address of the root node). For example you would like to interact with gateway C you could call it through A (the root of the tree): `http://ROOT_IP/c`. The root gateway will extract the request for c and forward transparently your call to c. Likewise you could call E by `http://ROOT_IP/c/e`.

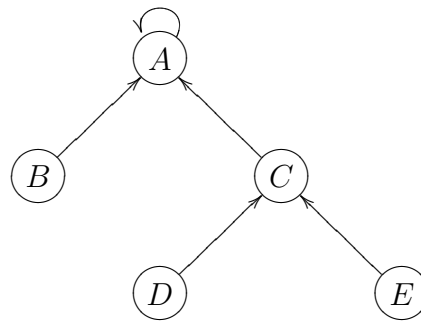


Figure 4.1: The figure shows a sample hierarchy of several gateways. Each gateway can possibly hold other devices as well.

4.2 Gateway registration

Each gateway has to know its parent. To register the child calls the "register" function at the parent by sending a POST-request containing the type of the child-gateway and the name of the gateway. The parent inspects the request and creates a "GatewayDriver" for the new child. If the driver registration works fine the parent sends a "ack" to the child, a "nack" otherwise.

The parent periodically performs a test on the driver to look if the child is still up and healthy. If this test fails the child gets removed from the list of available devices (Each child gateway acts also as a device comparable to eg. a sunspot attached directly to the parent). This ensures that failed/crashed clients get removed after some time.

4.2.1 Heartbeat

A child gateway periodically refreshes the "lease" at the parent by sending a heartbeat message. The heartbeat message contains several fields:

- **control** data like the name of the child etc...
- **keywords:** Semantic information about all the devices attached at the child (see 4.4.1).

If the parent does not recognize a child sending a heartbeat, the child gets informed about that with a "nack" message. The child then tries to register at the parent by calling the registration method. This allows the system to recover if the parent fails for just a few seconds.

4.2.2 Link recovery

One part of the link recovery protocol is already covered by the heartbeat mechanism. When a client cannot connect to its parent it retries in regular intervals. Upon failure the retry interval gets doubled (exponential back-off algorithm). Usually the parent should recover and then the clients will reregister.

However it is possible that a parent crashes forever and the tree never regenerates. To prevent such a tree fragmentation each client runs the backup-route protocol if the parent could not be reached for some time. Each client knows the root of the tree. It therefore contacts the root and asks the root to send the ip of the gateway with the longest matching prefix towards the client. The client then runs temporarily a registration process at this gateway.

See (Fig. 4.2): Gateway C has failed and therefore links DC and EC fail as well. Both gateways (D and E) now try to establish a backup-route to gateway A:

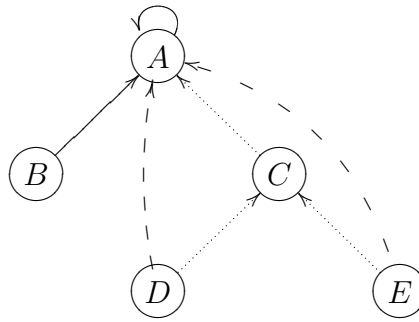


Figure 4.2: Link failure: Node C has crashed and therefore the links DC and EC are no more valid. Nodes C and E recover a backup-route to A

1. D sends the current location to the root. This is `http://ROOT_IP/c/d`
2. The root tries to route the request along the tree towards D (link AC then CD). As C cannot be contacted the root returns the ip address of A (in this case the root itself) to D.
3. D runs the registration process at A.

As soon as the correct route (over C) becomes available again, the backup-route gets dropped and the original route gets installed.

4.3 Device Listing and Description

On each gateway the client can request a listing of all available devices. Additionally all keywords of the child devices are listed.

When a client invokes the ip address or the uri of a gateway, the gateway will react by sending a list containing all the devices/gateways currently available.

Example:

`http://ROOT_IP/eastwing/floor3/` would return all the devices/gateways that are registered in floor3 together with the respective keywords (see 4.4.1).

If the client by accident or by purpose calls a device instead of a gateway then the device description gets generated and returned.

Example:

`http://ROOT_IP/eastwing/floor3/sensor1/` would return the capabilities and other informative stuff about the device sensor1.

4.4 Eventing

The eventing mechanism bases on the observer pattern. Clients can register to events and get notifications when the state of the resources has changed. The events are sent using HTTP to the event receivers.

4.4.1 Keywords for Semantic information

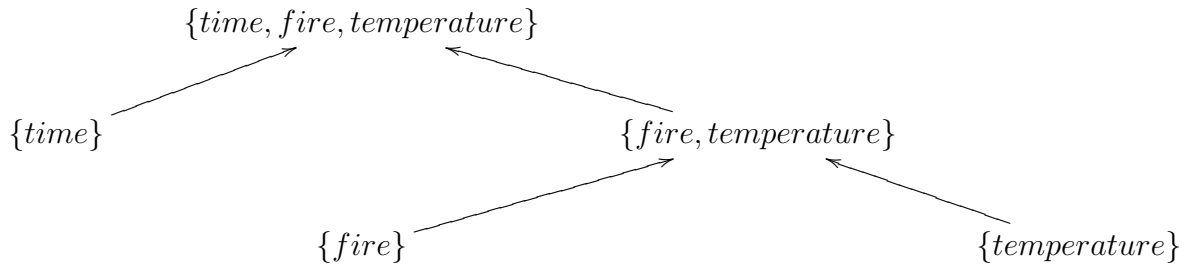


Figure 4.3: keyword propagation through the tree

To enrich sensors with some kind of semantic information all sensors get tagged with additional keywords that describe their capabilities. A firesensor would be enriched with the keywords fire and/or possibly smoke. A temperature sensor could allow keywords like heat, temperature or pressure. The gateway where the sensor is attached will collect those informations for all the attached sensors and will present them to the clients.

In addition all gateways propagate the keywords upwards towards the root taking the union of all the keywords from all their childs. This way all the gateways keep a repository of a formalized representation of the capabilities of at least one of the attached sensors in the subtree (Fig. 4.3).

4.4.2 Client registration and event refresh

To register for a certain type of event, the client sends an event registration message either directly to the responsible gateway or to the root. The message has to contain the following parameters:

- **keyword:** the keyword the event is listening on.
- **lease time:** the events lifetime until removed.
- **callback:** an uri where to send the event to if triggered.

A sample HTTP request with a lease time of 10 minutes interested into fire events would look like this:

```

POST /eastwing/floor3 HTTP/1.1
Host: ROOT_IP
Accept: application/rdf

keywords=fire
leaseTime=600
callback=http://sampleMe/callmeback
  
```

An event registration gets removed when its lease time is over. This is important as it might be possible that a client crashes and the gateway does not get informed about that incident. Then the event would eat up resources at the gateway.

To refresh a lease the client just has to send the same request again into the sytem.

4.4.3 Event installation

When a gateway receives an event registration message from a client it performs the following steps:

1. Lookup all the devices/gateways that provide the keyword requested.
2. Send an event registration message to each of this devices in case the device is a gateway. In case of a locally attached device, install the event in the eventing mechanism and start the timer that will remove the event when the lease time is over. If there is already an equal event registered just renew the lease time and restart the timer.

4.4.4 Event firing

The event firing is solved with the use of the observer pattern. In the observer pattern clients can register themselves at the observable object. If then the observable object has changed, it informs the client through the `setChanged()` method. Through the method `notifySubscribers(Object obj)` events can delivered containing the new information to the clients.

In the Smart Gateway architecture, a client can register itself over HTTP to a Device which implements the Observable interface. In the subscription form the client adds a call-back URI to specify where it can be reached. For receiving the events every client needs an `AsynchronousEventReceiver`, which listens on a specified port in order to receive the events fired by the Smart Gateways.

```
private void informChange() {
    setChanged();
    notifyObservers(new Event("fireControl", state, name, getContext()
        .getSymbolicLocation().getLocation()));
}
```

The above code snippet of a sample device driver shows the firing of an event at the Smart Gateway site.

Smart Simulator

"We grow too soon old and too late smart"

Dutch Proverb

This chapter describes the Smart Simulator and its components. As mentioned in chapter 3, we decided to base the simulator on JADE. First we describe the basic concepts of JADE and later the architecture of the simulator is explained.

5.1 JADE

JADE is a multi-agent simulation framework implemented in Java. The two main products of JADE are a fully FIPA¹ compatible agent-platform which simplifies the interaction with the multi-agent systems and a package to develop Java agents. Agent to agent communication is done by ACL messages. Every agent has its own private queue of received messages. Agents can access their queue by blocking, polling, timeout or pattern matching. In the following the main components of JADE are introduced: At first the agent platform and then some important concepts for developing agents with JADE.

5.1.1 Agent Platform

Each running instance of JADE is called a Container. The set of active containers is called a Platform. A single special Main Container must always exist in a platform, and all other containers have to register themselves to this container. In the following the important parts of the Main Container are introduced:

¹See <http://www.fipa.org> .

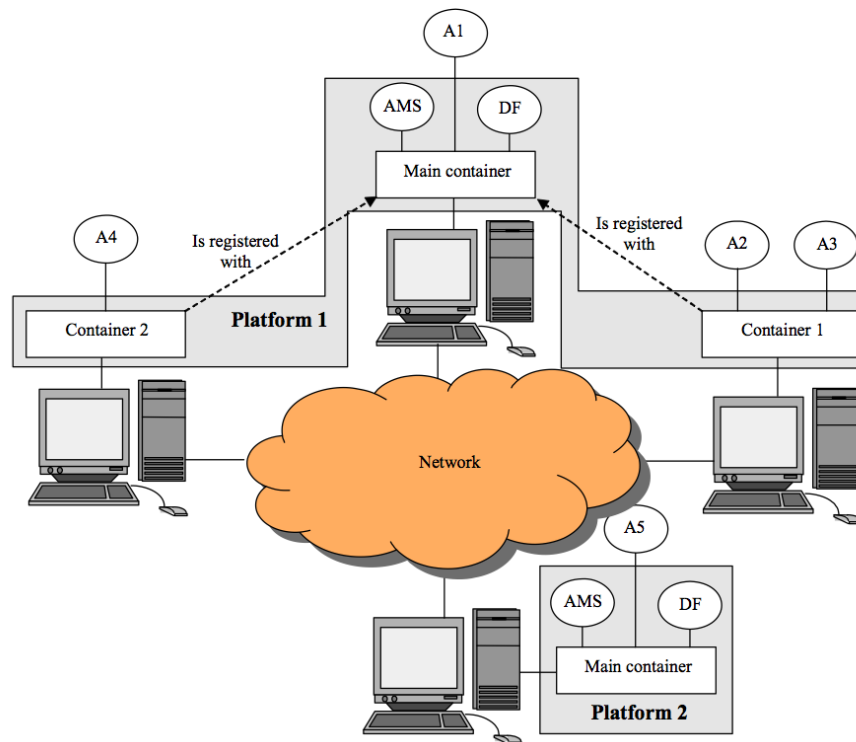


Figure 5.1: A sample JADE deployment with different Containers and Platforms. Source (Caire, 2007)

Agent Management System (AMS) This agent supervises the usage and access of the Agent Platform. For one platform there exists exactly one AMS. Each agent must register itself at a AMS in order to get valid. When registered, an agent receives an agent identifier (AID).

Directory Facilitator (DF) The DF provides the yellow pages service, where agents can register themselves or search other agents.

Message Transport System This system which is also called Agent Communication Channel (ACC) controls the message exchange (possibly from/to remote platforms) with the platform.

5.1.2 Agent

In JADE agents can be implemented as java class extending the JADE Agent class. This class has the abstract methods `setup()` and `takeDown()`, which are for initialization and termination respectively. In the `setup()` method behaviors can be added to the agent. A behavior defines the tasks of an agent.

5.1.3 Behavior

A behavior is a single-threaded routine of an agent. An agent can have multiple behaviors concurrently. Scheduling of behaviors is cooperative (not pre-emptive as for Java threads). If a behavior is executed, its method `action()` is called and runs until it returns.

In behaviors proactive and reactive actions can be defined. A typical proactive action is the sending of a message other agents. A reactive action is to react upon the receipt of a message. To allow such reactions efficiently behaviors can call a special `block()` method which stops the behavior until a certain message arrives.

To define a custom behavior for a user defined agent one has to implement a class which extends the abstract JADE Behavior class or any of its heirs. For a heir of the Behavior class the life cycle is determined by receiving messages and react upon them. A special heir, which is used as parent class besides the general Behavior class in the Smart Simulator is the class `TickerBehavior`. Heirs of this class have to implement the abstract method `onTick()`, which gets executed periodically. This behavior template was used for simulating sensing activities of a simulated device. In our implementation the device executes a method called `getNewState()`, which simulates a new sensing activity.

5.1.4 Communication

As mentioned above the DF facilitates a central service where agents can find the AIDs of other agents. Using such AIDs, agents can send messages to the message queues of other agents. Agent to agent messages are sent in the Agent Communication Language (ACL). For ACL messages the type (such as REQUEST, INFORM, etc.), the receiver(s), and a conversation ID can be defined. Agents can send and receive such messages.

5.2 Simulator components

The simulator consists of different interacting parts. The part which a normal user gets to see is the Simulator Controller user interface. This GUI allows the user to generate a random tree structure of several Smart Gateways instances by specifying their position in the tree as well as their hosts and ports. Further at any position simulated devices can be added as a child to any gateway. When the Smart Simulator starts up all registered device types are added to a list of devices displayed by the GUI.

5.2.1 Simulation Controller

The simulation controller consists of two parts:

- **Simulation Controller Agent** This agent invokes the GUI and receives messages from other agents.
- **Simulation Controller UI** The graphical GUI of the Smart Simulator.

Simulation Controller UI This GUI allows the user to generate a random tree structure of several Smart Gateways instances by specifying their position in the tree as well as their hosts and ports. When a gateway is entered into the tree, the URI of the parent gateway is set accordingly in the newly created instance. The instance is created by a `ServerWrapper` class, which is able to start and stop Smart Gateway instances. Further at any position simulated devices can be added as a child to any gateway. When a new device was created, it receives the parent gateway URI as an argument. A simulated device corresponds to a JADE based device agent. For each newly created device, an agent simulating such a device agent is added and started in the Smart Simulator. Using the received parameters the device agent communicates

with the parent gateway through a RESTlet client. When the Smart Simulator starts up all registered device types are added to a list of devices from which the user can choose.

The GUI contains two `JTabbedPane`s. The first Pane represents the Smart Gateway structure. Here all instances of the Smart Gateways which are created inside the Smart Simulator are displayed in a `JTree`. Also the attached simulated devices are shown in this tab. In the other tab, all registered Internal Devices are listed. As they do not have a specific location in the tree, the Internal Devices are listed in a separate `JList`, not part of the Smart Gateway structure.

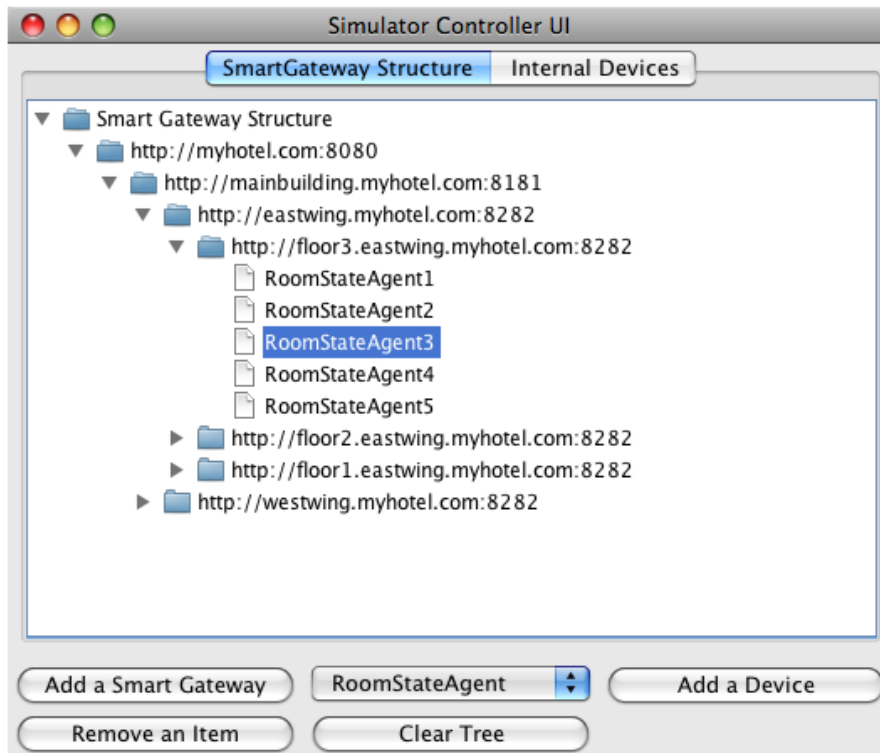


Figure 5.2: The Simulation Controller UI allows users to generate a Smart Gateway structure and append simulated devices to the gateways.

Simulaton Controller Agent This agent starts the Simulator Controller UI. Further all internal devices (can) register themselves to the UI through the Simulation Controller agent. For the current state of the simulator, the registered internal devices are only registered in the `JList` of the Simulation Controller UI. But as an easy extension it would be possible to add parameters for stopping them, or to start specific internal agents.

5.2.2 Device Agents

Simulated devices are divided into internal and normal devices. Internal devices do not communicate outside the simulator. As they have no specific location and are not attached to a gateway they are not part of the Smart Gateway structure created for the simulation. Normal devices are attached to a gateway and have a fixed and specific place inside the Smart Gateway. They communicate with their parent gateway.

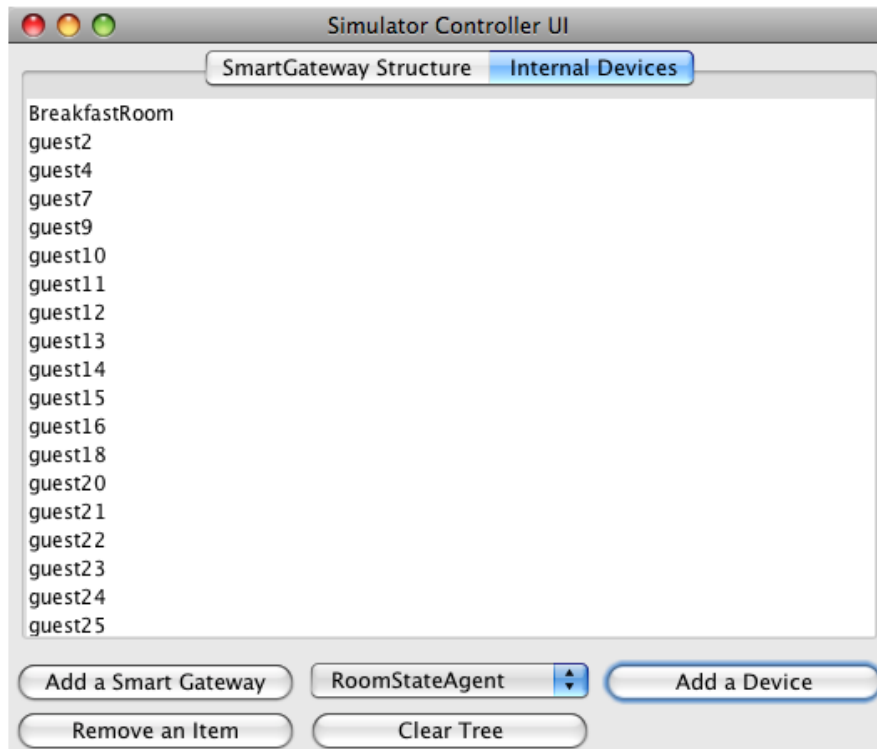


Figure 5.3: The second tab of the Simulation Controller UI shows the registered internal devices.

For creating devices there are three abstract classes which provide basic functionalities for simulated devices:

- **Abstract Device Agent** Abstract super class for all simulated devices.
- **Internal Device Agent** Abstract super class for internal devices.
- **Device Agent** Abstract super class for device agents.

Abstract Device Agent This abstract class collects some common functionalities for devices:

- **Agent Registration** All devices are registered at the JADE yellow pages in order for other agents to find and communicate them.
- **Search Single Agent** This method searches a single agent in the JADE yellow pages and returns its AID.
- **Send to Internal Agent** This method sends a message to a receiver inside the JADE environment.

Internal Device Agent The abstract class `InternalDeviceAgent` inherits from the class `AbstractDeviceAgent`. Further it provides a method for the registration of an internal devices at the simulation controller. It is not mandatory for an internal device to register itself at the simulation controller, it is however a good idea to do it in order to have an overview of all the internal devices inside a simulation.

At the moment, only the name of the internal device is sent to the simulation controller. For obtaining the ability of stopping certain internal devices inside the simulation controller, as a second argument the AID of the internal device agent could be passed. Using this identifier the simulation controller could terminate the agents as done for the device agents.

Device Agent This abstract class collects common methods for simulated devices which are attached to a Smart Gateway. Basically it provides the communication with the parent gateway. This is done by a RESTlet client which sends HTTP requests to the gateway. When a device agent is created, then a corresponding driver needs to be instantiated at its parent gateway. This is done by sending a PUT request to the gateway. This request contains the name of the device and a fully qualified name of the driver class for the simulated device. The figure 5.4 shows this method inside the `DeviceAgent` class.

To update a device state a POST request is sent to the parent gateway. This works similar as the before mentioned creation request. In the update request the name of the device, the new state of the device and the keyword of the device are sent to the gateway. The driver at the parent gateway is then responsible for updating the device representation accordingly.

For the implementation of a specific device agent the user has to implement a class which extends the class `DeviceAgent`. Then behaviors have to be added to the agent in order to define the simulated tasks of the agent. How to implement an appropriate behavior for such an agent is explained in section 5.2.4.

5.2.3 Internal Agents

Internal agents are simulated devices which do not communicate with outside actors. The purpose of internal agents is to define special devices which do not need to have a specific location in the hierarchical Smart Gateway structure. They communicate with other internal devices and with simulated devices. Internal devices can register themselves at the `SimulationControllerAgent`. For the implementation of an internal agent the abstract class `InternalDeviceAgent` has to be extended.

5.2.4 Simulation Behavior

To define the tasks of a simulated device, one or more behaviors have to be added to the device agent. The classes `SimulationBehavior` and `SimulationTickerBehavior` provide templates for such behaviors. The difference of these two behaviors is that `SimulationTickerBehavior` represents a proactive behavior, where periodically a state transition of the simulated device is performed. The class `SimulationBehavior` provides a template for a reactive behavior. Both templates supply a method `updateState()` which updates the state of the simulated device.

For a heir of the class `SimulationTickerBehavior` the only method which has to be implemented is `getNewState()`. This method simulates a new state transaction of the simulated device and if the state has changed, then the new state of the device updated. The idea is that every simulated device is a state machine, which has a discrete number of states and a probability distribution for state changes.

5.3 Smart Simulator Application Scenarios

For testing, debugging, but also for a development of an application there exist different types of simulations for the Smart Simulator. The following section shall introduce the different

```

/**
 * Try to create a new Device Reference.
 *
 * @param name
 *         the name of the device
 * @param qualifiedClassName
 *         the qualified name of the class in the Smart Gateway framework
 *         implementing the required Device.
 */
public void createDeviceReference(String name,
                                 String qualifiedClassName) {

    if (parentHostUri == null) {
        // The URI of the resource "list of items".
        parentHostUri = (String) getArguments()[0];
    }
    // Gathering informations into a Web form.
    Form form = new Form();
    form.add("name", name);
    form.add("class", qualifiedClassName);
    Representation rep = form.getWebRepresentation();
    Reference ref = new Reference(parentHostUri
                                 + "/_createDev");

    // Launch the request
    Response response = client.put(ref, rep);
    if (response.getStatus().isSuccess()) {
        if (response.isEntityAvailable()) {
            try {
                // Always consume the response's entity, if available.
                response.getEntity().write(System.out);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}

```

Figure 5.4: A pure Scenario. Only simulated devices are part of the application.

simulations and point out what benefits the different configurations bring along.

5.3.1 Pure Simulation

For a rapid development of a scenario the pure simulation configuration will be the best solution. With this configuration all devices are simulated using the Smart Simulator. The hierarchical Smart Gateway structure is built using the Smart Simulator Simulation Controller. Figure 5.5 shows such a scenario.

Some of the devices are attached to the instantiated Smart Gateways and others are internal devices inside the simulator. The gateways correspond to the real Smart Gateways, except that all simulated gateways are located on the same host where the simulation is running.

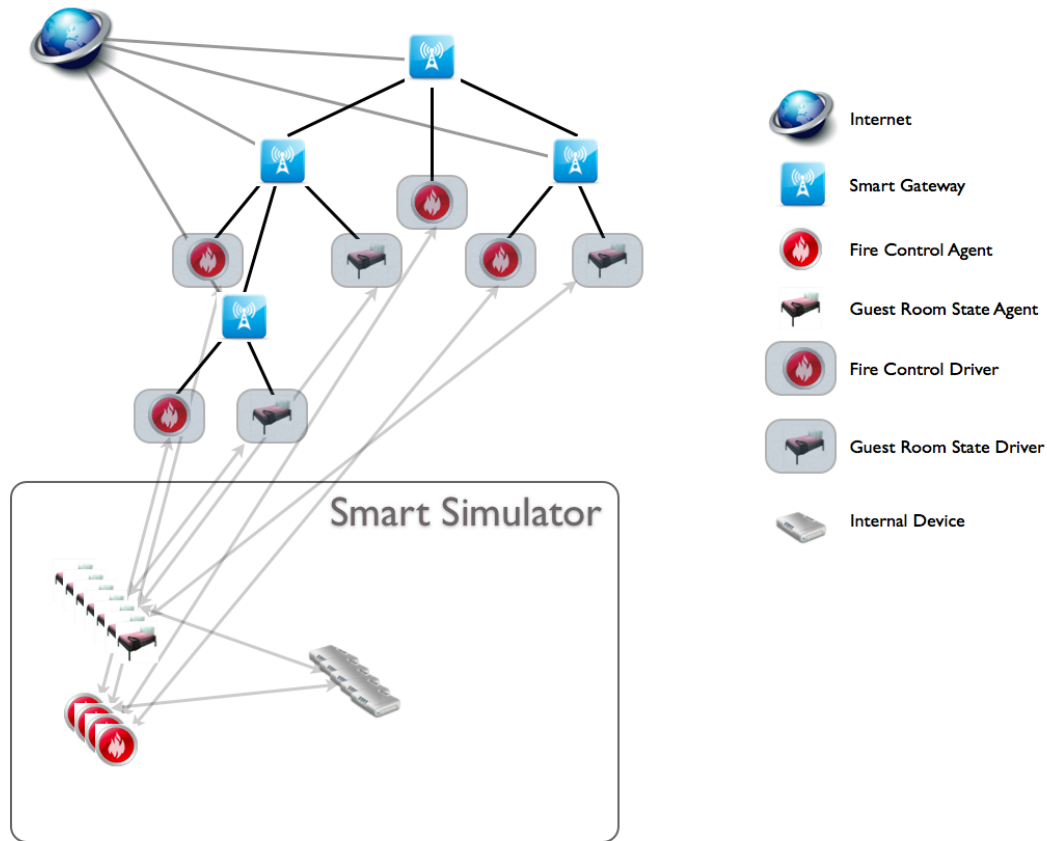


Figure 5.5: A pure simulated Scenario. Only simulated devices are part of the application.

5.3.2 Hybrid Simulation

In a hybrid setting the Smart Simulator is integrated into an existing structure of Smart Gateways with possibly, but not necessarily real devices attached to it. Additionally to simulated devices which are attached to a gateway, also internal devices can be defined which communicate with the simulated devices. The integration of the Smart Simulator is very flexible. It is possible to add simulated devices to a gateway where already real devices are attached to. Alternatively it is possible to create some mock up gateways with (or without) attached simulated devices attached to. These mockup gateways can act as children of real Smart Gateways or as their parents. Figure 5.6 shows such a scenario.

A possible scenario for a hybrid scenario is to extend existing hardware devices with simulated ones. The simulated devices act as a state machine. The transition probabilities can be defined for the simulated devices after some measurements with the actual hardware devices. Using this scenario with only few hardware devices a huge simulated environment can be implemented.

5.3.3 Deployed Simulation

In a deployed simulation only real devices are part of the application. No internal devices can be added. Instances of Smart Gateways can be added to the Smart Gateway structure. Such a deployment can be done in a testing or in an actual operation setting. Figure 5.7 shows

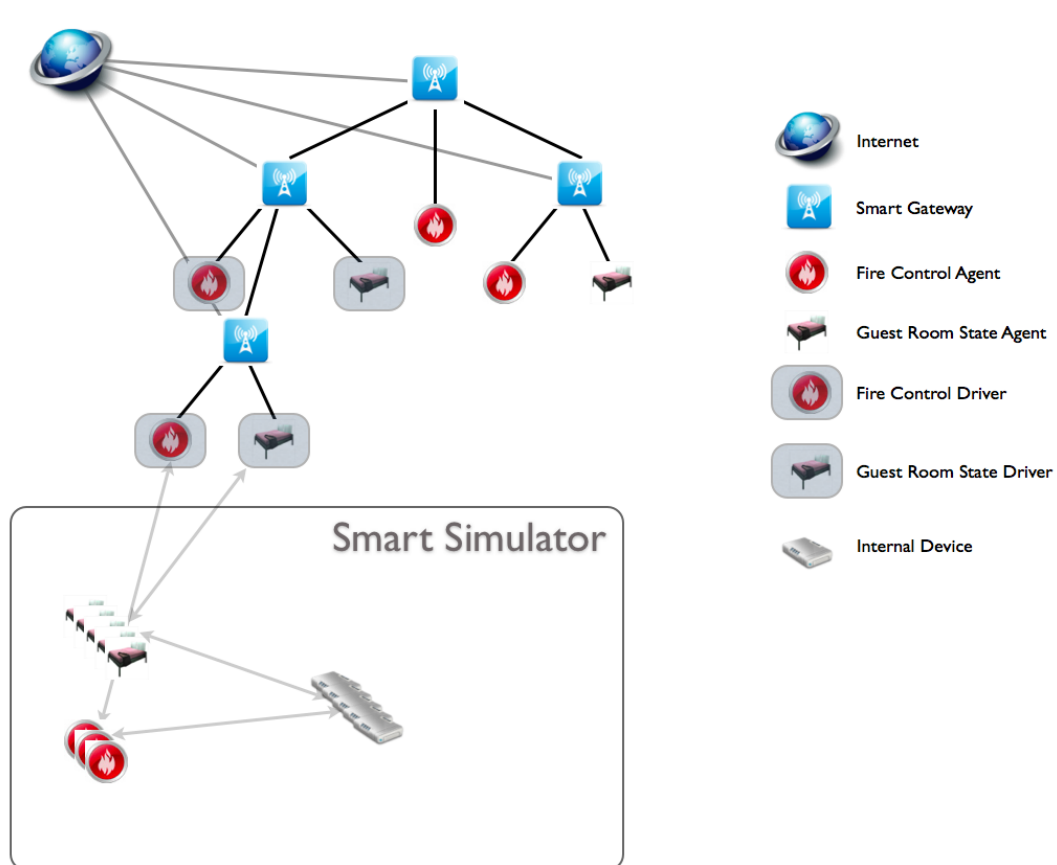


Figure 5.6: A hybrid simulation Scenario. Both real devices and Smart Gateways as well as simulated ones are part of the application.

such a scenario where a part of a building application is deployed, and some Smart Gateways are instantiated using the Smart Simulator.

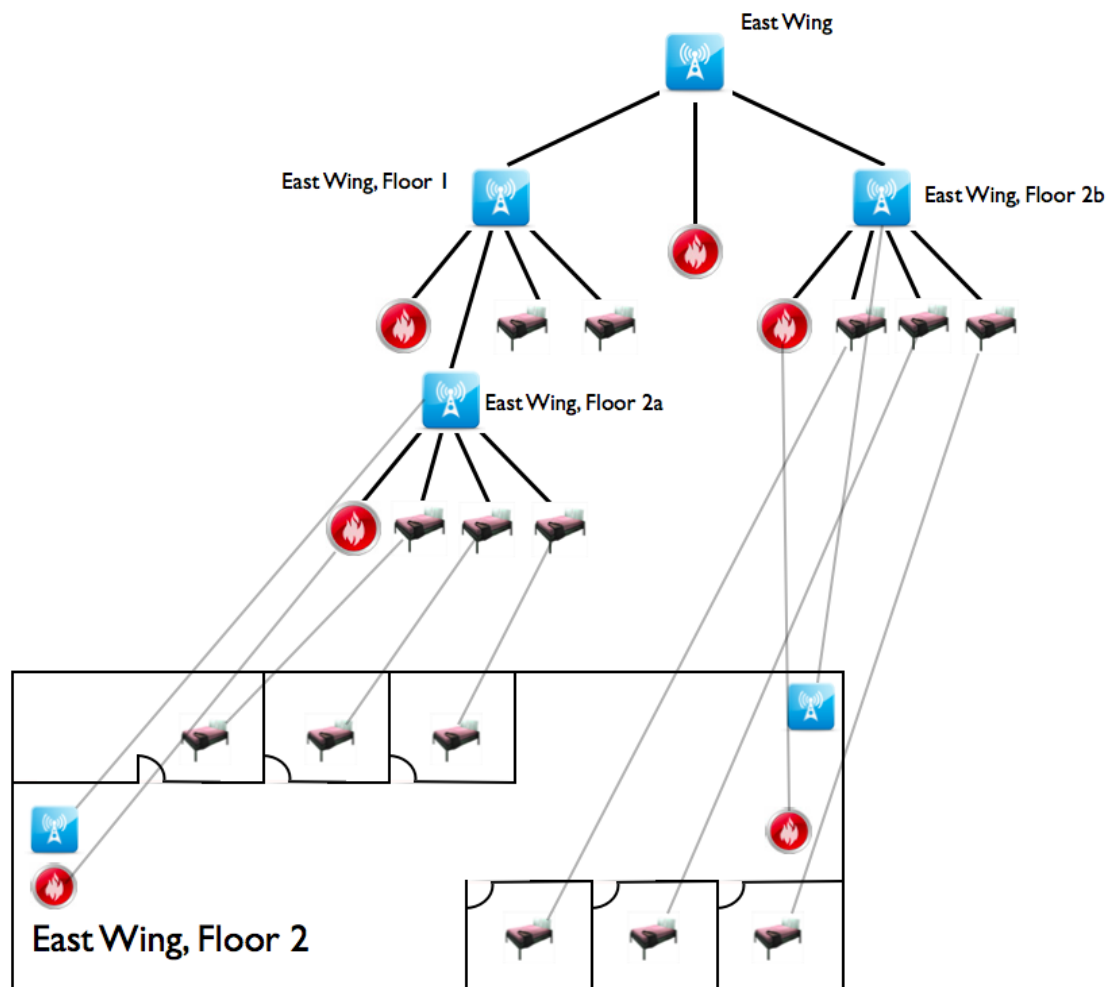


Figure 5.7: A deployed application Scenario. Only real devices and Smart Gateways are part of the application.

Chapter 6

External Client

”[The] ideal client is the very wealthy man in very great trouble.”

John Sterling (1806-1844)

We divide this chapter into different client categories which will be useful for different applications or client tasks. First, the simplest client is a simple *Web browser* which provides a well known interface for a user to browse through a Smart Gateway structure. Second, there has to be an *administration tool* for the building and maintenance of such a structure. And thirdly, a tool which might be useful for a specific application is an *external client application*.

6.1 Browser

As all the communication between Smart Gateways is done using HTTP a simple Web browser allows a user to explore a hierarchical Smart Gateway tree structure. Each Smart Gateway has a HTML representation of itself where the connected devices and child gateways are listed. A user can retrieve informations about an attached device by just clicking on its link. If the device is a gateway it self then the representation of this gateway is returned. If the device is a local device, then a HTML representation of this device is return which is generated by the gateway itself. The figures [6.1](#), [6.2](#) and [6.3](#) show example interactions of a Web browser.

6.2 Administration Tool

When deploying a Smart Gateway structure in a real environment with real sensor devices which attach to the structure, then there raises the need of an administration tool which allows a system administrator to add and delete Smart Gateways and devices to the structure. The Smart Simulator provides such a tool for the simulator purpose and could also be used for

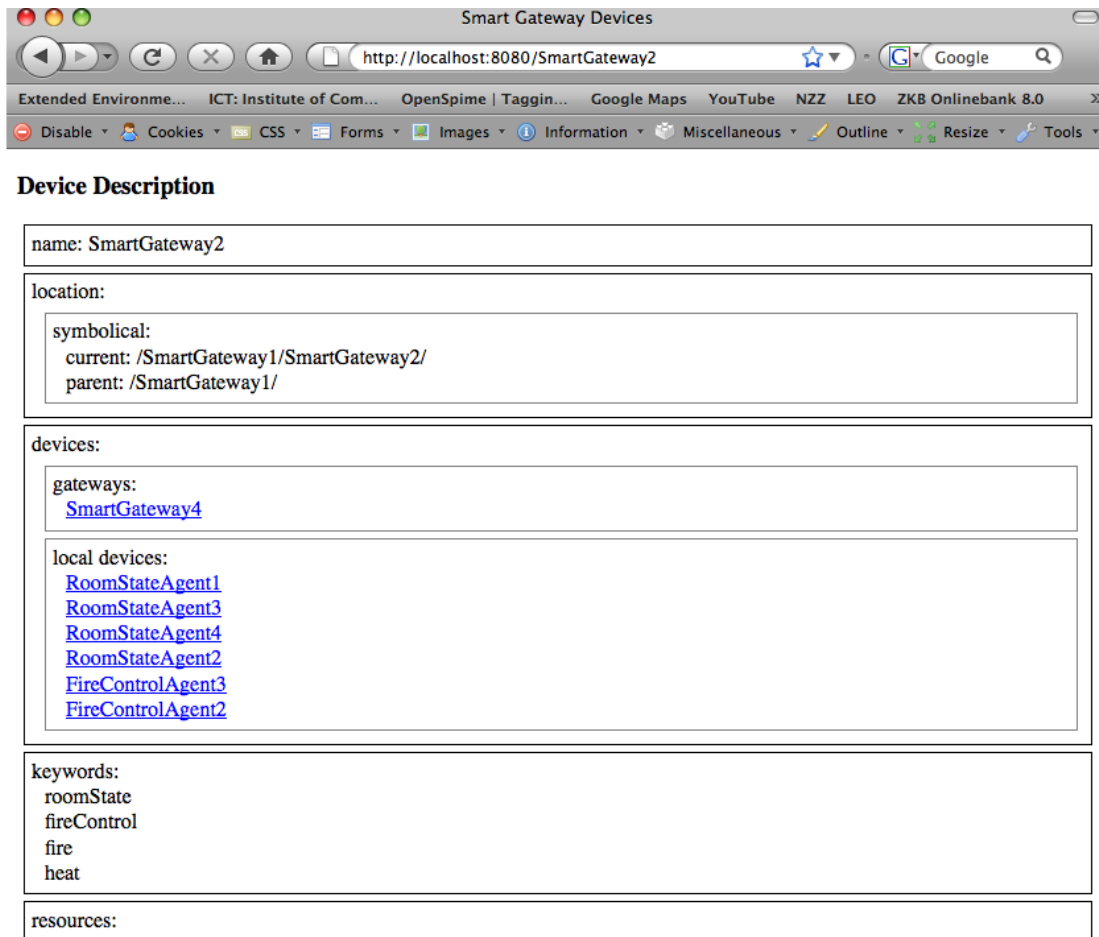


Figure 6.1: A HTML representation of a Smart Gateway with attached devices and one child gateway.

the real deployment. However for a better separation of the concepts it might be desirable to develop an separate tool for building and maintain such a structure.

6.3 External Client Application

Here we examine the concept and the architecture of the External Client. The client can be used to subscribe to certain events. For subscribing to a certain event the following parameters must be chosen:

- **host** The host where the remote Smart Gateway is located can be found. Clients need to select the top Smart Gateway from which they want to receive events.
- **port** The port where the remote Smart Gateway is listening.
- **keyword** The keyword specifies what events the client is interested in.
- **leasetime** This parameter specifies the time the client is interested in these events. After the mentioned leasetime the Smart Gateways will stop to send events to this client.

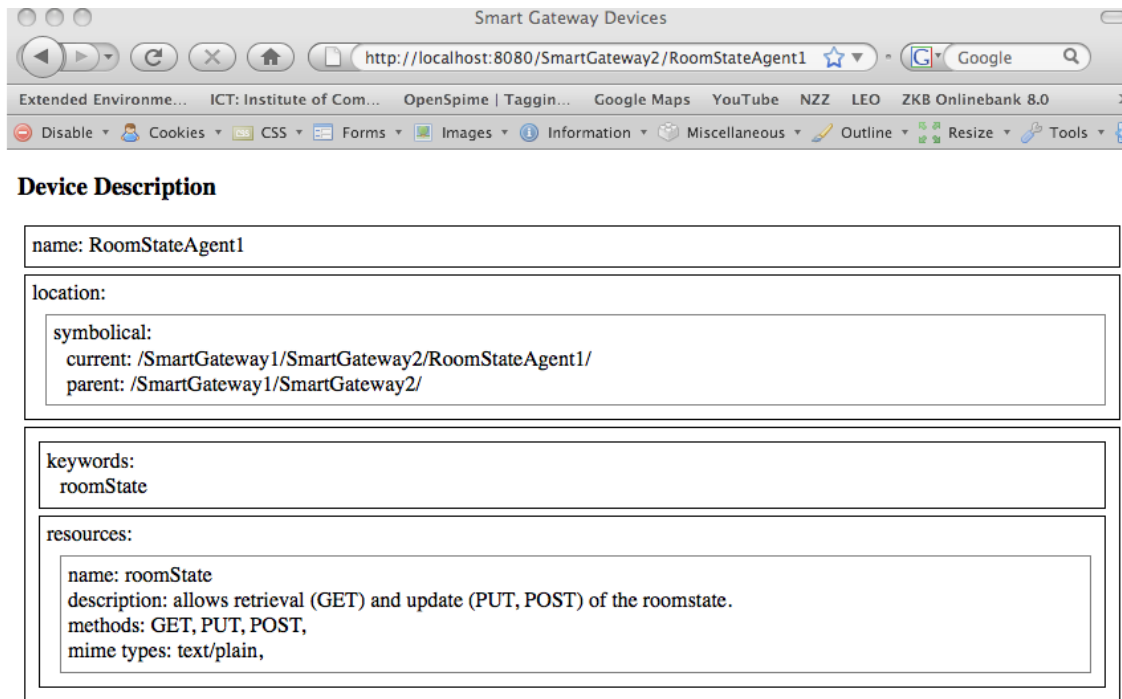


Figure 6.2: A HTML device representation. This representation is generated from the gateway.

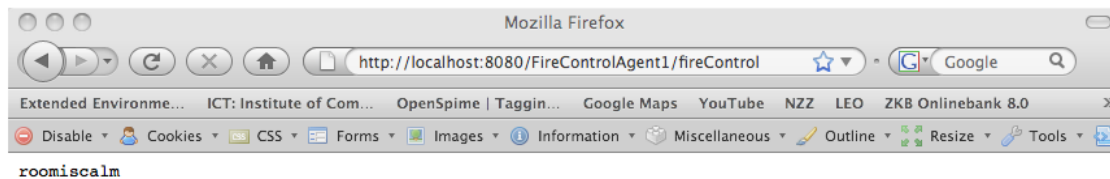


Figure 6.3: A textual representation of a device state. The device is attached to a gateway.

- **callback** The callback is the URL where the client can be reached. This client's determined automatically by the External Client.

After subscribing to events, the external client will receive events of the specified type.

The External Client is a demonstrator how easily client applications can be developed from the Smart Gateway framework. From the aim of a rapid prototyping for ubicomp scenarios the External Client brings the basic functionality for a non computer scientist to interact with the simulated Smart Gateway structure and the simulated or existing devices attached to the gateways. For our sample scenario application we have implemented a external client application which can be used as standalone java application or alternatively as java applet. Please see chapter 7 for further informations on the sample application.

6.3.1 Architecture

The external client exists as a standalone java application and as a java applet. It consists of a board, which communicates with the Smart Gateway host and receives events from it. A

general user interface from which a user can subscribe to certain events and set specific states to specified devices. Finally every device whose events are received by the board is represented by a special panel, which corresponds to the device type.

In the following the main components are introduced.

Board The external client board has both a RESTlet client and server. The RESTlet client takes care of the communication with the foreign host of the remote Smart Gateway structure. The RESTlet server receives the events which are sent by the foreign host. An event consists of a device name, the device location, the value measured by the device and a keyword specifying the type of the measurement. The board then handles the received events and either adds a new panel to the external client UI in case the event was the first received from the corresponding device, or updates the panel in case the board has received events before from this device.

UI The external client UI is a generic user interface which allows the user to subscribe to certain, through a keyword specified events. For this the user has to specify the foreign host and port together with the keyword. The leasetime specifies how long the user will receive the requested events. Once events are received, and displayed in the UI, the client can set specific states for a device by sending a "set state" request using the UI.

Device Panel A device panel is a graphical representation of a device and its state for the user which appears in the UI. The panel consists of a name and a state which represent the device name and its current state. For the generic external client panel, the state is displayed as text. Specific panels can place certain icons for specific states.

Sample Application

”Do Not Disturb” signs should be written in the language of the hotel maids”

Tim Bedore (1968-)

Here we present the sample application we implemented. The reason to build it was to have a running application with all components working together in order to identify missing pieces or failures of our framework and to demonstrate a possible scenario where our framework would provide interesting benefits. Later in the chapter we give an explanation of how to implement your own sample application.

7.1 Hotel Scenario

Every hotel tries to create the illusion for every guest that the room which the guest occupies completely belongs to the guest. The guest is thereby given a place of absolute privacy away from the own home. But this illusion is difficult to maintain, when every day the room has to be cleaned and maintained by the house keeping. A person working for housekeeping is often confronted with the unpleasant task to knock at a guest room in order to find out whether the guest is currently in his room. If no answers comes, the employee still has to consider the possibilities that the guest has not heard the inquiry from the housekeeping employee. The employee can only guess whether the guest is still in the room or not. On the other hand the inquiry from the employee disturbs the guest as his privacy gets violated. ”Do not disturb” or ”Please make room now” signs only weaken this problem slightly, but do not really solve it.

This reoccurring example was the motivation for our sample application. In the following the actors of the example will be introduced. The devices which compose our sample application are equipped with capabilities which are technically feasible with todays technology.

7.1.1 Smart Gateway Structure

A hotel has one or more buildings, which have several floors. Distributed in the building(s) there are guest rooms and special rooms like breakfast rooms, the entrance lobby or the reception. For our simulation we map such a generic hotel building structure to Smart Gateways at this mentioned locations. The idea is that every place with a semantic meaning for the hotel has a Smart Gateway to which all devices of that place are connected. The connection between the Smart Gateways is according to the semantic of the building. If for example a building would have a east wing with tree floors and ten rooms on every floor, then the Smart Gateway of the floor number 3 in the east wing would have the Smart Gateway of the east wing as parent, and all Smart Gateways of all the rooms located at the floor 3 as children.

7.1.2 Devices

The hotel scenario includes a set of different devices. Some are internal devices and do not communicate with the Smart Gateway structure. All non internal devices are attached to the Smart Gateway structure and report their states to the Smart Gateway where they are attached to. The state is only reported if it is different to the last sent state.

RoomState Device This device are responsible for monitoring the state of a guest room. Every guest room has one RoomState device. A RoomState device knows if it is currently occupied by a guest or vacant. If occupied by a guest it registrates itself to the BreakfastRoom to enter and exit events of its guest. If occupied by a guest the device periodically senses the state of the room and sends its state to the parent gateway. The device receives alerts from the BreakfastRoom and changes its state accordingly: If an enter alert arives from the BreakfastRoom, then the device changes its state from "roomIsEmpty" to "guestIsBreakfasting". If a exit alert arrives from the BreakfastRoom it changes its state from "guestIsBreakfasting" to "roomIsEmpty".

FireControl Device This device is a fire detection device which monitors a room, floor or building region. This device has tree different states. "roomIsCalm" means that there is no fire or smoke in the monitored region. "roomIsSmoking" means, that smoke is detected, but no fire, whereas "roomIsOnFire" for a detected fire stands.

The actors of this applications are hypothetical roomstate devices which simulate and determine the state of a guest room. this devices get attached to smart gateways which are located at compontens such as floor, building of the hotels. a room knows whether it has a guest or whether its not occupied; if the room is occupied by a guest it subscribes at the breakfast room for enter and exit events of the specified guest; thereby the room can detect besides guestinroom and noguest inroom also guestisbreakfasting.

BreakfastRoom The BreakfastRoom represents a device located in the breakfast room of the hotel. The function of the BreakfastRoom device ist to detect guests which enter or leave the breakfast room. If a room has subscribed for events for a specific guest, then the breakfast room sends these informations in case of an event.



Figure 7.1: The different states of the RoomState device

7.1.3 External Client

For our implemented scenario we have two external clients which profit from the provided services.

Housekeeping Employee A member of the housekeeping staff can subscribe to room state events of the floor of the current position. Obtaining all the room states of the rooms located at the same floor as the employee, the employee can schedule its activity according to the received informations. Further to keep track of the completed work and as information for the other employees, the room maid can set a "cleaning" or "clean" state to the rooms which are currently being cleaned or have been cleaned.

Security Employee A member of the security can subscribe to all fire control events of the part of the hotel which is under his surveillance. For more informations of peoples location in case of fire in a part of the hotel he can also subscribe to room state events. In case of fire the security employee can activate the sprinkling equipment of the affected and nearby region of the hotel.

7.2 Create your own Application

The Smart Simulator aims to be the ideal base for deploying a ubicomp scenario in a short time with only little configuration and implementation effort. In the following the needed tasks



Figure 7.2: The different states of the FireControl device

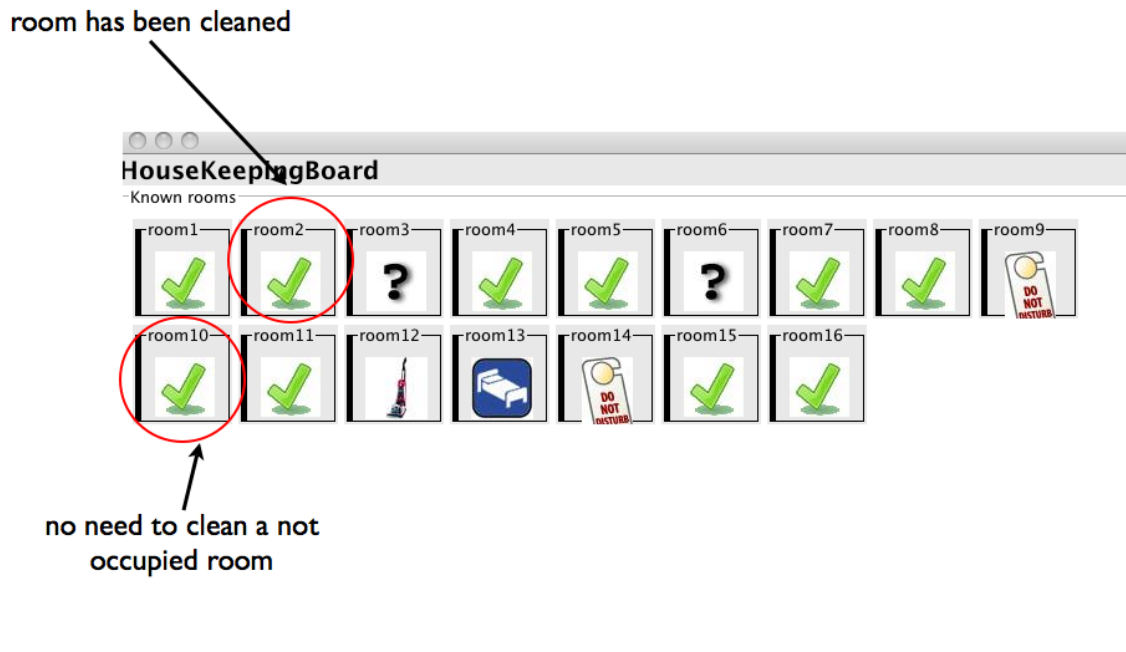


Figure 7.3: A housekeeping board with the received states of all roomState devices located in the same floor as the housekeeping employee.

for implementing such a scenario will be described. For implementing an own application the Smart Simulator as well as the Smart Gateway project have to be extended.

7.2.1 Smart Simulator

In the simulator simulated devices can be created. The creation of a new device consists of four parts:

- **Device Object** Define a object class for your device.
- **Device Agent** Implement a device agent, as base for your simulated device.
- **Device Behavior** Define one or more behaviors for your device.
- **Internal Device Agents** Optional: if needed for the simulation of your scenario you can define internal device agents.

The device agent is the actor of the simulated device. It holds a device object where the state and the properties of the simulated device are present. The agent adds one or more behaviors, which simulate the device activities. If needed there are other intern devices, which communicate with the device agent.

Device Object The device object class represents the simulated device as a Java object. The device agent holds such an object for keeping track of the current properties of the simulated device. The class `SimulationDevice` represents a device with fields of a generic device: The `deviceName`, the keyword of the device, and the device state. Either these fields are enough for your simulated device, or, in case you want to define more fields, you can implement a own device object class. This class would have to inherit from the class `SimulationDevice`. See Appendix C for an example of such an object.

Device Agent The central part in order to define a new simulated device is to write a class which inherits the abstract class `DeviceAgent`. Let's call this class `MySampleDeviceAgent`. For a sample implementation of this class have a look at `FireControlAgent`. This new class must not be abstract, but implement two abstract methods of the parent class. This class is then the agent of the simulated device which communicates with the gateway where it is attached to. Apart from the communication the device agent is responsible for the initialization and the termination of the simulated device. See APPENDIX A for an example of such an implementation.

Three tasks have to be executed in the initialization: The device object and the RESTlet client initialization, and the addition of behaviors to the device agent.

When `MySampleDeviceAgent` is created and started at first the method `setup()` is executed. In this method performs all the three mentioned initialization steps. First you call the method `getDeviceConfiguration()`. In this method which you have to implement by your own, the device object is instantiated. Second, you have to call the method `initializeSettings()`. In this method the RESTlet client for the communication with the parent gateway is instantiated. In order to achieve this you have to extract the host of the parent gateway URI from the class arguments. Then you call the parent method `initRESTletClient(String, String)` with the device name, and the class name of the implementing handler at the gateway side. The last task is to add behaviors to the device agent. The behavior is added to the agent as last step in the agent initialization. The added behavior(s) specify actions of the device.

Device Behavior A behavior specifies the actions which are carried out by the simulated device during its lifetime. For defining a device behavior a new class has to be implemented, let's call it `MyDeviceBehavior`, which inherits from the abstract class `SimulationBehavior`. This later class is an extension of the JADE behavior class `Tickerbehavior`, In the `SimulationBehavior` class the method `onTick()` gets executed periodically. This method calls the abstract `getNewState()` method, which returns a boolean value specifying whether the state of the device has changed. If the state has changed, then the new state is communicated to the parent gateway using the device agent.

`MyDeviceBehavior` has only to implement the abstract `getNewState()` method of its parent class. So the class is merely a state machine, which calculates the new state. If the current state of the device has to be taken into account for the new device state, then this state can be obtained, by `myAgent.getState()`. See Appendix B for an example implementation of such a behavior.

Internal Device Agent If needed internal devices can be added to the scenario which communicate among them and with the simulated devices. Implementing an internal device is quite similar like the implementation of a normal device. There is no specific need to device a device object for internal devices, it can however be done. To implement an internal device, a new class has to be written which inherits from the class `InternalDeviceAgent`.

As there is no RESTlet client for an internal device, initialization of the agent consists only of two parts. First the device has to be initialized, then the internal device has to be registered at the simulation controller. This can be done using the provided method of the parent class `registerInternalDeviceAtControlGUI(String deviceName)`.

Finally a behavior can be added to the internal device agent in order to define its activities.

7.2.2 Smart Gateway

For every device defined in the simulator, there has to be an appropriate handler at the Smart Gateway side. For this reason a class have to be defined which extends the abstract class `Device` and implements the interface `Pollable`. The class `Device` defines the template of a sample handler class. The interface `Pollable` defines the ability to being polled for devices which rarely change their states. The most important method is the `handle()` method. In this method the client request is handled and the new state is set. The sent parameters have to be extracted from the sent request. See Appendix E for a sample implementation.

7.2.3 External Client

As we have mentioned in chapter 6, there exist three kinds of clients: First, a regular Web browser, second an administration tool for building and maintaining the Smart Gateway structure, and third a external client application which is basically just a application specific interface for registering and receiving events and to set certain states to designated devices.

While for the first two clients, a new application will not change the existing tools, the third client, the external client application can be adapted for a newly created scenario. For this reason a class has to be created which implements the abstract class `ExternalClientPanel`. For a sample implementation of this interface see the Appendix D. The class has to define icons for the different states of the device. This icons will display the current state of the device in the client board. For every event received from a device satisfying the requested keyword and location, the external client UI creates a panel and includes this to the UI. Events of devices already included in the UI will update the panel with the newly received state of it.

Conclusions and Future Work

"My interest is in the future because I am going to spend the rest of my life there."

Charles F. Kettering (1876-1958)

Here we summarize our work and identify possible future directions of enhancements or extensions for it.

8.1 Future Work

Enhancements and extensions are possible in a variety of directions. Due to the modular design of our framework extensions can be integrated with a minimum of changes to the framework.

8.1.1 Security

In the implemented framework no security enforcing mechanisms are present. Keeping typical applications in mind they are located at highly security critical locations of a company. So equipping our framework with security will be inevitable for real applications. Basically three kinds of security will be needed for securing our framework:

- **Access Control** In real application environments access control will be of crucial importance. Secure mechanisms will have to decide and enforce which user will be given access to what resources. Access restrictions must be implemented for both external and internal users.
- **Secure Communication** For the protection of both confidential business and personal data secure communication encryption will be needed.

- **Authentication** The authentication of an entity would be needed to decide whether a entity can be given a permission for a certain action.

Access Control A very successful approach towards the challenges of access control is the role based access control. Thinking of our sample applications there are clearly distinguishable roles of users which can be given specific access permissions. For example a guest may be given access for its own room and all floors gateways. On the other hand a housekeeping employee must be given access to all room gateways.

Secure Communication As the whole communication between a Smart Gateway and other Smart Gateways or an attached device is done using HTTP it comes as a positive side effect, that changing from HTTP to HTTPS could bring relatively easily an encrypted communication without the need of major changes in the architecture.

Authentication The authentication needs to be addressed separately. It will be important that an entity which claims to have a certain role represents indeed the proper entity.

8.1.2 Use case studies

As the Smart Simulator and the Smart Gateway framework aim to bring a benefit to companies which could deploy a homemade application located in a scenario they would like to test a user case study could be a very interesting survey. To implement at large scale the sample hotel scenario presented in chapter 7 with the collaboration of a existing hotel in terms of building structure and statistical room state data for accurate transition probabilities would provide an impressive showcase and an interesting testing environment for both the Smart Simulator as well as for the Smart Gateway structure.

8.1.3 And now for something completely different...

To extend our existing showcase example has surely a large potential to discover strengths and weaknesses of our framework. Jet another way to discover them would be a completely different application scenario. Here are some sketches of possible scenario located in a completely different direction:

A fire-detection system for large forests Every year we hear from tragic forrest fires in different regions of the earth. If a fire breaks out in a large forrest under certain climatic conditions it can't be distinguished anymore, but must in a struggle of multiple days or weeks be controlled and stemmed. Apart from the costs which reach billions of dollars every year, thousands of people loose their homes to the flames.

The only possible way to become hand over the fire earlier would be to detect the fire much earlier. A scenario application could simulate and investigate such a pervasive system bringing such a benefit. A theoretical solution could be the spreading of transmitting devices in such forests. The distribution would be for a area of several sq.km. With the density of the devices would be in a such manner, that every device could communicate with at least n (e.g. $n=5$) devices. some smart gateways which have a higher cost that the smart dust devices could be distributed as a backbone frame. If devices detect fire they communicate it (via other devices) to the nearest smart gateway. This gateway would route it to the fire control base station.

The devices would have to be able to communicate through a radio media and they would have to have the ability to detect fire, for instance by measuring a temperature higher than 70 deg Celsius. The construction of such device is technically feasible, but there would have to be an estimation about the cost per device. These costs would be opposed to the potential saving of a much earlier fire detection.

Devices could be defined with certain capabilities and then tests could be simulated of in what time the outbreak of a fire could be detected at a firefighting base station and with what error probability. These simulation could estimate the technical requirements for such devices relatively precisely, so that a sample testbed could be developed to compare predicted and actual performance.

Failure in the real application would have to be taken into account in the whole simulation. Forest is not a ideal place for communication, and during the distribution of the devices some of them could get broke. Failures like these would be needed to be included in the simulation in order to obtain a realistic simulation result.

Such a scenario would have multiple scientific benefits for our framework as result:

- **Scalability** This scenario could be simulated with a configuration of thousands of devices. Such a simulation would discover unsparing scalability or performance problems.
- **Different Types of Simulation comes with one code** For such a scenario, at first a pure simulation could be implemented. After that a prototyping hardware simulation could be combined with simulated devices to a hybrid simulation. In the hybrid simulation prototype built from SunSPOTs or BTnodes could be used in order to substitute the hypothetic not existing devices. In such a hybrid simulation with hardware prototyping devices set in a real forest new technical challenges like error-prone communication and could be analyzed. With the obtained simulation result the data sent by the hardware devices could be compared with the data sent from the simulated devices in order to improve the simulated device behaviors. Further precise requirements for real devices could be specified. Using this specification, a producer of embedded devices could present the cost of their development in mass-production. These costs could be compared with potential hopefully higher savings through the use of the system.
- **Open Space Implementation** The Smart Gateways include a hierarchical concept of location. In buildings this concept is surely adequate. In such a scenario it could be analyzed if such a concept is adequate for the use in open space.

8.1.4 Ontologies

As we have seen in chapter 2 there has to be a solid base in order that the use of ontologies become advantageous. We think to have built such a base with our device representation ontology. Building upon this ontology, semantic informations could be included into the device representation with the use of techniques such as RDFa or microformat. As a result there would be semantic informations integrated in the device representation. Such information could help the devices to share context informations among them and react on context changes. This leads to more autonomy in their behavior and therefore reduces the configuration effort for users.

8.2 Conclusions

In our project we have presented the Smart Gateway architecture as a resource-oriented approach for the integration of embedded devices into the Web. Our modular architecture allows the adaption of a broad variety of devices. The Smart Gateway produces for each attached device a web representation of it. We have defined a device representation ontology in order to formally define the representation of a device. As our framework follows the Web standards and interacts through HTTP, a simple web browser is suitable as a client. Our framework includes a simulator which allows to define hypothetical devices with hypothetical capabilities. These devices can be attached to a Smart Gateway. Thereby it is possible to interact with them using a web browser. The main aim of the Smart Simulator is the rapid development of pervasive computing simulation scenarios. The simulator is able to instantiate a hierarchical structure of Smart Gateway instances. Using this hierarchy, locations can be defined in a hierarchical manner in the simulated scenarios. Using the Smart Simulator different types of simulations are possible: The first type is a pure simulation where all devices and the whole gateway structure is defined by the simulator. A second possibility is a hybrid simulation where a part of the gateway structure and/or a part of the devices can be real deployed instances. The integration of the simulator into an existing Smart Gateway structure or the attachment of real devices to gateway instances which were defined using the simulator is very flexible and requires only little configuration effort. A third type of simulation is a scenario where the only task of the simulator is the instantiation of some gateways in order to connect existing gateways or to attach real devices to it.

The vision of the "Web of Things" where every day objects are integrated into the Web requires a capable and scalable architectural base. Our framework shows a promising approach in this direction and concentrates on embedded devices and allows extensions in different directions. As seen in the section 8.1 there exist many interesting starting points of possible extensions and enhancements of our work.

Appendix A : Sample Simulation Device Agent

This section shows the abstract class `DeviceAgent` and a sample extension of it.

Abstract Class `DeviceAgent`

```
package ch.ethz.inf.vs.gateway.sim.agents;

import jade.wrapper.AgentController;
import jade.wrapper.StaleProxyException;

import java.io.IOException;

import org.restlet.Client;
import org.restlet.data.Form;
import org.restlet.data.Protocol;
import org.restlet.data.Reference;
import org.restlet.data.Response;
import org.restlet.resource.Representation;

import ch.ethz.inf.vs.gateway.sim.helpers.SimulationDevice;

/**
 * This agent provides general device capabilities which are reused by its
 * ancestors.
 *
 * @author haennimi
 */
public abstract class DeviceAgent extends
    AbsrtactDeviceAgent {

    /** The RESTlet client to communicate with the parent gateway. */
    protected Client client;

    protected SimulationDevice device;

    public SimulationDevice getDevice() {
        return device;
    }

    public void setDevice(SimulationDevice device) {
        this.device = device;
    }

    /**
```

```

* The host of the parent gateway. This parameter comes as an argument
* during agent initialization.
*/
protected String parentHostUri;

/**
* Try to create a new Device Reference.
*
* @param name
*         the name of the device
* @param qualifiedClassName
*         the qualified name of the class in the Smart Gateway framework
*         implementing the required Device.
*
*/
public void createDeviceReference(String name,
    String qualifiedClassName) {

    if (parentHostUri == null) {
        // The URI of the resource "list of items".
        parentHostUri = (String) getArguments()[0];
    }
    // Gathering informations into a Web form.
    Form form = new Form();
    form.add("name", name);
    form.add("class", qualifiedClassName);
    Representation rep = form.getWebRepresentation();
    Reference ref = new Reference(parentHostUri
        + "/_createDev");

    // Launch the request
    Response response = client.put(ref, rep);
    if (response.getStatus().isSuccess()) {
        if (response.isEntityAvailable()) {
            try {
                // Always consume the response's entity, if available.
                response.getEntity().write(System.out);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

}

/**
* Try to update the device state.
*
* @param name
*         the name of the device.
* @param action
*         the new state of the device.
* @param keyword
*         the keyword of the device.
*/
public void updateDeviceState(String name,
    String action, String keyword) {
    // Gathering informations into a Web form.

```

```

Form form = new Form();
form.add("state", action);
Representation rep = form.getWebRepresentation();

// Launch the request
Response response = client.post(parentHostUri + "/"
    + name + "/" + keyword, rep);
if (response.isEntityAvailable()) {
    try {
        // Always consume the response's entity, if available.
        response.getEntity().write(System.out);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

}

/** The guest staying in this room needs to be created. */
public void createInternalAgent(String agentName,
    String className, Object[] args) {
    AgentController guestAgent;
    try {
        guestAgent = getContainerController()
            .createNewAgent(agentName, className,
                args);

        guestAgent.start();
    } catch (StaleProxyException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

public void initRESTletClient(String deviceName,
    String className) {

    // Define our Restlet HTTP client.
    client = new Client(Protocol.HTTP);

    createDeviceReference(deviceName, className);
}

/** Put agent initializations here. */
public abstract void initializeSettings();

/** Put device initialization here. */
protected abstract void getDeviceConfiguration();
}
}

```

Sample Simulation Device Agent

```

package ch.ethz.inf.vs.gateway.sim.fireControl;

import ch.ethz.inf.vs.gateway.sim.agents.DeviceAgent;

/**
 * The firecontrol agent. The firecontrol simulates firecontrol events and
 * forwards them to the parent Smart Gateway.

```

```

*
* @author haennimi
*/

public class FireControlAgent extends DeviceAgent {

    /**
     * The qualified class name of the class in the Smart Gateway framework
     * implementing the driver for this device.
     */
    private static String CLASS_NAME =
        "ch.ethz.inf.vs.gateway.plugin.devices.drivers.fireControl.FireControl";

    /** The FireControlRoomState which keeps track of the state of this agent. */
    private FireControlDevice fireControlDevice;

    /** Put agents initializations here */
    protected void setup() {

        // receive Room essentials from simulation controller
        getDeviceConfiguration();

        initializeSettings();

        // Register the room in the yellow pages
        register(fireControlDevice.getDeviceName(), "fireControlAgent");

        // Print a welcome message
        System.out.println("hello, " + fireControlDevice.getDeviceName()
            + " is ready.");

        addBehaviour(new FireControlBehavior(this, 3000));
    }

    @Override
    public void initializeSettings() {

        Object[] args = this.getArguments();

        // The URI of the resource "list of items".
        parentHostUri = (String) args[0];

        initRESTletClient(fireControlDevice.getDeviceName(), CLASS_NAME);
    }

    protected void getDeviceConfiguration() {
        fireControlDevice = new FireControlDevice(getLocalName());
        this.device = fireControlDevice;
    }

    // Put agent clean-up operations here
    protected void takeDown() {
        // Printout a dismissal message
        System.out.println("room " + fireControlDevice.getDeviceName()
            + " is terminating.");
    }
}
}

```

Appendix B : Sample Device Behavior

This section shows a sample behavior implementation which extends the behavior template of the class `SimulationTickerBehavior`.

```
package ch.ethz.inf.vs.gateway.sim.fireControl;

import jade.core.Agent;
import ch.ethz.inf.vs.gateway.sim.agents.DeviceAgent;
import ch.ethz.inf.vs.gateway.sim.behaviours.SimulationTickerBehavior;

public class FireControlBehavior extends SimulationTickerBehavior {

    /** Barrier for new events */
    public float newEventBarrier = (float) 0.7;

    /** Chance that room begins to smoke */
    private static double SMOKING_CHANCE = 0.03;

    /** Chance that smoke in room changes to fire */
    private static double SMOKE_CHANGES_TO_FIRE = 0.1;

    public FireControlBehavior(Agent a, long period) {
        super(a, period);
        this.agent = (DeviceAgent) a;
    }

    protected boolean getNewState() {
        FireControlDevice device = (FireControlDevice) agent.getDevice();
        if (device.isOnFire()) {
            newEventBarrier = (float) 0.99;
        } else {
            newEventBarrier = (float) 0.7;
        }
        if (Math.random() > newEventBarrier) {
            System.out.println("Generating new fire controll events for room "
                + device.getDeviceName());

            // Perform the simulation
            newState();
            return true;
        } else {
            return false;
        }
    }

    private void newState() {
```

```
double d = Math.random();
String state = "roomIsCalm";
FireControlDevice room = (FireControlDevice) agent.getDevice();
if (!(room.isOnFire() && room.isSmoking())) {
    if (d > SMOKING_CHANCE && !room.isSmoking()) {
        state = "roomIsCalm";
    } else if (d < SMOKING_CHANCE && !room.isSmoking()) {
        room.setSmoking(true);
        state = "roomIsSmoking";
    } else if (room.isSmoking() && d < SMOKE_CHANGES_TO_FIRE) {
        room.setOnFire(true);
        state = "roomIsOnFire";
    }
} else {
    if (d > 0.9) {
        room.setOnFire(false);
        room.setSmoking(false);
    } else if (room.isOnFire()) {
        state = "roomIsOnFire";
    } else {
        state = "roomIsSmoking";
    }
}

agent.getDevice().setState(state);
}
```

Appendix C : Sample Simulation Device

This section shows a sample device object of a simulated device. It inherits from the class `SimulationDevice`.

```
package ch.ethz.inf.vs.gateway.sim.fireControl;

import ch.ethz.inf.vs.gateway.sim.helpers.SimulationDevice;

public class FireControlDevice extends SimulationDevice {

    private static String KEYWORD = "fireControl";

    // The current state of the device
    private String roomState = "unknown";

    // Is there smoke in the room?
    private boolean isSmoking = false;

    // Is there fire in the room?
    private boolean isOnFire = false;

    public FireControlDevice(String roomName) {
        super(KEYWORD, roomName);
    }

    public String getRoomState() {
        return roomState;
    }

    public void setRoomState(String roomState) {
        this.roomState = roomState;
    }

    public boolean isSmoking() {
        return isSmoking;
    }

    public void setSmoking(boolean isSmoking) {
        this.isSmoking = isSmoking;
    }

    public boolean isOnFire() {
        return isOnFire;
    }

    public void setOnFire(boolean isOnFire) {
```

```
        this.isOnFire = isOnFire;
    }
}
```

Appendix D : Sample External Client Panel

This section shows the abstract class `ExternalClientPanel` and a sample extension of it.

Abstract Class External Client Panel

```
package ch.ethz.inf.vs.gateway.sim.externalclient.gui;

import java.awt.EventQueue;
import java.net.URL;

import javax.swing.ImageIcon;
import javax.swing.JPanel;

import ch.ethz.inf.vs.gateway.sim.externalclient.outside.housekeeping.HouseKeepingPanel;

/**
 * The ExternalClientPanel The external client panel displays small icons as
 * state representatives on the ExternalClientUI. A concrete implementation has
 * to inherit from this class and provide a 80x80 pixel icon for every state the
 * device can obtain. A inheriting class must implement the abstract method
 * update() where the icon must be set according to the present state of the
 * device. The icons have to be initialized in the constructor. For a reference
 * implementation of such a inheriting class see {@link HouseKeepingPanel}.
 *
 * @author haennimi
 *
 * */
public abstract class ExternalClientPanel extends JPanel {

    protected ExternalClientBoard myBoard;
    protected String state;

    public abstract void update(String state);

    public ExternalClientBoard getMyBoard() {
        return myBoard;
    }

    public void setMyBoard(ExternalClientBoard myBoard) {
        this.myBoard = myBoard;
    }
}
```

```

public String getState() {
    return state;
}

public void setState(String state) {
    this.state = state;
    update(state);
}

protected ImageIcon createImageIcon(String path, String description) {
    ImageIcon icon = null;

    URL imgURL = getClass().getResource(path);
    if (null != imgURL) {
        icon = new ImageIcon(imgURL, description);
    }
    if (icon == null) {
        System.out.println("no image found at " + path);
    }
    return icon;
}

protected void doIt() {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            ExternalClientPanel.this.validate();
            ExternalClientPanel.this.repaint();
        }
    });
}
}

```

Sample Client Application Panel

```

package ch.ethz.inf.vs.gateway.sim.externalclient.outside.firecontrol;

import java.awt.Color;
import java.awt.Dimension;

import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.border.MatteBorder;
import javax.swing.border.TitledBorder;

import ch.ethz.inf.vs.gateway.sim.externalclient.gui.ExternalClientBoard;
import ch.ethz.inf.vs.gateway.sim.externalclient.gui.ExternalClientPanel;

public class FireControlPanel extends ExternalClientPanel {

    public ExternalClientBoard myBoard;
    public String state;
    private ImageIcon stateUnknownIcon;
    private ImageIcon stateSmokingIcon;
    private ImageIcon stateOnFireIcon;
    private ImageIcon stateOKIcon;
    private ImageIcon stateSprinklingIcon;
}

```

```
private JLabel stateUnknown;
private JLabel stateSmoking;
private JLabel stateOnFire;
private JLabel stateOK;
private JLabel stateSprinkling;

public FireControlPanel(Object fireControlBoard, String title) {
    super();
    this.myBoard = (ExternalClientBoard) fireControlBoard;

    Dimension dim = new Dimension(80, 80);
    setSize(dim);
    setMaximumSize(dim);
    setMinimumSize(dim);
    setPreferredSize(dim);
    setBorder(new TitledBorder(new MatteBorder(1, 5, 1, 1, Color.BLACK),
        title));
    stateUnknownIcon = createImageIcon(
        "/ch/ethz/inf/vs/gateway/sim/externalclient/resources/unknown.jpeg",
        "State");
    stateUnknown = new JLabel(new String(), stateUnknownIcon,
        SwingConstants.LEADING);

    stateOnFireIcon = createImageIcon(
        "/ch/ethz/inf/vs/gateway/sim/externalclient/resources/fire.gif",
        "State");
    stateOnFire = new JLabel(new String(), stateOnFireIcon,
        SwingConstants.LEADING);

    stateSmokingIcon = createImageIcon(
        "/ch/ethz/inf/vs/gateway/sim/externalclient/resources/smoke.gif",
        "State");
    stateSmoking = new JLabel(new String(), stateSmokingIcon,
        SwingConstants.LEADING);

    stateOKIcon = createImageIcon(
        "/ch/ethz/inf/vs/gateway/sim/externalclient/resources/clean.jpeg",
        "State");
    stateOK = new JLabel(new String(), stateOKIcon, SwingConstants.LEADING);

    stateSprinklingIcon = createImageIcon(
        "/ch/ethz/inf/vs/gateway/sim/externalclient/resources/sprinkler.jpg",
        "State");
    stateSprinkling = new JLabel(new String(), stateSprinklingIcon,
        SwingConstants.LEADING);

    this.add(stateUnknown);
    this.setVisible(true);
}

public void update(String state) {
    if (state.equalsIgnoreCase("roomIsOnFire")) {
        this.removeAll();
        this.add(stateOnFire);
    } else if (state.equalsIgnoreCase("roomIsSmoking")) {
        this.removeAll();
        this.add(stateSmoking);
    } else if (state.equalsIgnoreCase("roomIsCalm")) {
        this.removeAll();
    }
}
```

```
        this.add(stateOK);
    } else if (state.equalsIgnoreCase("sprinkling")) {
        this.removeAll();
        this.add(stateSprinkling);
    } else {
        this.removeAll();
        this.add(stateUnknown);
    }
    doIt();
}
}
```

Appendix E : Sample Smart Gateway Driver

This section shows the parts of the handler structure of the Smart Gateway. For every simulated device in the Smart Simulator there has to be a handler defined inside the Smart Gateway framework, which implements the interface `Pollable` and extends the class `Device`. The two classes and a sample implementation of such a driver are listed below.

Interface Pollable

```
package ch.ethz.inf.vs.gateway.api.poll;

/**
 * allows some item to be polled by the polling service.
 * @author sawielan
 *
 */
public interface Pollable {

    /**
     * on poll will be called in regular intervals by the polling service.
     * @param time the time when the onPoll was invoked.
     */
    public void onPoll(long time);
}
```

Abstract Class Device

```
package ch.ethz.inf.vs.gateway.plugin.devices;

import java.util.Map;
import java.util.Observable;

import org.restlet.data.Request;
import org.restlet.data.Response;

import ch.ethz.inf.vs.gateway.plugin.devices.info.Context;
import ch.ethz.inf.vs.gateway.plugin.devices.info.DeviceDescription;
import ch.ethz.inf.vs.gateway.plugin.devices.info.Resources;
import ch.ethz.inf.vs.gateway.plugin.devices.info.XMLRepresentable;

/**
 * interface for a device stored in the system.
 * @author sawielan
 */
```

```

*
*/
public abstract class Device extends Observable implements XMLRepresentable {

    /** the context of this device. */
    protected Context context = new Context();

    /** the resources description for this device. */
    protected Resources resources = new Resources();

    /** the device description for this device. */
    protected DeviceDescription deviceDescription = new DeviceDescription();

    /** flags whether cache is ok or not. */
    protected boolean xmlCacheDirty = true;

    /** the cache string. */
    protected String xmlCacheString = null;

    // ABSTRACT METHODS TO BE IMPLEMENTED

    public abstract void init(Map<String, Object> params);

    /**
     * returns the device name. within the gateway the device name has to be
     * unique.
     * @return the device name.
     */
    public abstract String getDeviceName();

    /**
     * sets the name of the device. be careful with this method. especially do
     * not call it when you have already installed the device in the gateway.
     * otherwise this could lead to inconsistency to the device mapping within
     * the gateway.
     * @param deviceName the new name for this device.
     */
    public abstract void setDeviceName(String deviceName);

    /**
     * handle a request.
     * @param response the response to the request.
     * @param request the request that was performed on the gateway.
     * @return the representation as String or if set in response null.
     */
    public abstract String handle(Response response, Request request);

    /**
     * test flag to look if this device is alive.
     * @return true if alive, false otherwise.
     */
    public abstract boolean isAlive();

    /**
     * sets a device alive. the device itself periodically informs the driver
     * if it is still alive.
     */
    public abstract void setAlive();

```

```
// END ABSTRACT METHODS TO BE IMPLEMENTED

/**
 * returns a handle to the context of this device.
 * @return the context of this device.
 */
public Context getContext() {
    return context;
}

/**
 * gives a handle to the resources description this device is providing.
 * @return the resources description provided by this device.
 */
public Resources getResources() {
    return resources;
}

/**
 * gives a handle to the description of this device.
 * @return the description of this device (model, vendor, ...).
 */
public DeviceDescription getDeviceDescription() {
    return deviceDescription;
}

public void setXMLCacheDirty(boolean dirty) {
    this.xmlCacheDirty = dirty;
}

public boolean isXmlCacheOk() {
    if (xmlCacheDirty) {
        return false;
    }
    if (!context.isXmlCacheOk()) {
        return false;
    }
    if (!resources.isXmlCacheOk()) {
        return false;
    }
    return true;
}

public String asXML() {
    if (isXmlCacheOk()) {
        return xmlCacheString;
    }

    StringBuffer buf = new StringBuffer();
    buf.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
    buf.append("<device>");

    buf.append("<name>" + getDeviceName() + "</name>");

    // append the context
    buf.append(getContext().asXML());
}
```

```

        // append the service description
        buf.append(getResources().asXML());

        // append device description
        //str.append(getDeviceDescription().asXML(false));

        buf.append("</device>");
        xmlCacheString = buf.toString();
        setXMLCacheDirty(false);
        return xmlCacheString;
    }
}

```

Sample Gateway Handler

```

package ch.ethz.inf.vs.gateway.plugin.devices.drivers.fireControl;

import java.util.List;
import java.util.Map;

import org.apache.log4j.Logger;
import org.restlet.data.Form;
import org.restlet.data.MediaType;
import org.restlet.data.Method;
import org.restlet.data.Parameter;
import org.restlet.data.Request;
import org.restlet.data.Response;
import org.restlet.data.Status;

import ch.ethz.inf.vs.gateway.Core;
import ch.ethz.inf.vs.gateway.api.poll.Pollable;
import ch.ethz.inf.vs.gateway.plugin.devices.Device;
import ch.ethz.inf.vs.gateway.plugin.devices.info.Resource;
import ch.ethz.inf.vs.gateway.plugin.eventing.Event;

/**
 * models a simulator of a fire control sensor.
 *
 * @author sawielan, haennimi
 *
 */
public class FireControl extends Device implements Pollable {

    /** log4j instance. */
    private static Logger log = Logger.getLogger(FireControl.class);

    /** the name of the state parameter in the request. */
    public static final String PARAM_NAME = "state";

    /** the name of this fire control device. */
    private String name = null;

    /** the status of the fire control. */
    private String state = FireControlStates.UNKNOWN;

    /** the name of the fireControl rest command. */
    public static final String METHOD_FIRECONTROL = "fireControl";

    /** Chance that guest is in room. */

```

```

public static final double SMOKING_CHANCE = 0.03;

/** Chance that smoke changes to fire. */
public static double SMOKE_CHANGES_TO_FIRE = 0.1;

/**
 * constructor for a new room state object.
 *
 * @param name
 *         the name of the new object.
 */
public FireControl(String name) {
    this.name = name;

    getResources()
        .addResource(
            new Resource("fireControl", new Method[] { Method.GET,
                Method.PUT, Method.POST },
                new MediaType[] { MediaType.TEXT_PLAIN },
                "allows retrieval (GET) and update (PUT, POST)
                of the firecontrol.");

    getContext().getKeywords().add("fireControl");
    getContext().getKeywords().add("fire");
    getContext().getKeywords().add("heat");
}

@Override
public String getDeviceName() {
    return name;
}

@Override
public String handle(Response response, Request request) {
    List<String> segments = request.getResourceRef().getSegments();
    // don't proceed if device name not matching...
    if (!name.equals(segments.get(0))) {
        return asXML();
    }
    // remove the device name
    segments.remove(0);

    if (segments.size() > 0) {
        // there are more segments to be processed
        String m = segments.get(0);

        if (METHOD_FIRECONTROL.equals(m)) {
            fireControl(segments, response, request);
            return null;
        }
    }

    return asXML();
}

@Override
public void init(Map<String, Object> params) {
    Core.getInstance().getPollingService().register(this, 500);
}

```

```

}

@Override
public boolean isAlive() {
    return true;
}

@Override
public void setAlive() {
    // do nothing.
}

@Override
public void setDeviceName(String deviceName) {
    this.name = deviceName;
}

/**
 * sets the fire control state.
 *
 * @param segments
 *         the segments of the request.
 * @param response
 *         the response.
 * @param request
 *         the request.
 */
private void fireControl(List<String> segments, Response response,
    Request request) {

    org.restlet.data.Method m = request.getMethod();
    log.debug("executing fireControl");
    if (Method.POST.equals(m) || Method.PUT.equals(m)) {
        // update/put new value
        Form form = request.getEntityAsForm();
        Parameter parm = form.getFirst(PARAM_NAME);
        if (null == parm) {
            log.error("parameter missing");
            response.setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
        } else {
            String newState = parm.getValue();
            // only set the state if the new state is a valid state.
            if (FireControlStates.contains(newState)) {
                if (!state.equals(newState)) {
                    state = newState.toLowerCase();
                    informChange();
                    log.debug("change state");
                }
            }
            response.setEntity(state, MediaType.TEXT_PLAIN);
        }
    } else if (Method.GET.equals(m)) {
        // return the current room state.
        response.setEntity(state, MediaType.TEXT_PLAIN);
    }
}

public boolean onFire() {

```

```

        return FireControlStates.ROOM_IS_ON_FIRE.equals(state);
    }

    public boolean isSmoking() {
        return FireControlStates.ROOM_IS_SMOKING.equals(state);
    }

    private void informChange() {
        setChanged();
        notifyObservers(new Event("fireControl", state, name, getContext()
            .getSymbolicLocation().getLocation()));
    }

    public void onPoll(long time) {
        informChange();
    }
}

```

Device States

hier kommt dann der Quellcode...

```

package ch.ethz.inf.vs.gateway.plugin.devices.drivers.fireControl;

public class FireControlStates {
    public static final String ROOM_IS_CALM = "roomIsCalm";
    public static final String ROOM_IS_SMOKING = "roomIsSmoking";
    public static final String ROOM_IS_ON_FIRE = "roomIsOnFire";
    public static final String SPRINKLING = "sprinkling";
    public static final String UNKNOWN = "unknown";

    public static final String[] states =
        new String[] {
            ROOM_IS_CALM,
            ROOM_IS_SMOKING,
            ROOM_IS_ON_FIRE,
            SPRINKLING,
            UNKNOWN
        };

    /**
     * tests if a given state is contained.
     * @param state the state to test.
     * @return true if state contained, false otherwise.
     */
    public static boolean contains(String state) {
        for (String s : states) {
            if (s.equals(state)) {
                return true;
            }
        }
        return false;
    }
}

```


Bibliography

- Aberer, K., Hauswirth, M., and Salehi, A. (2007). Infrastructure for data processing in Large-Scale interconnected sensor networks. In *Mobile Data Management, 2007 International Conference on*, pages 198–205.
- Beeharee, A. and Steed, A. (2007). Exploiting real world knowledge in ubiquitous applications. *Personal Ubiquitous Comput.*, **11**(6), 429 – 437.
- Berners-Lee, T., Fielding, R., and Masinter, L. (2005). RFC 3986, Uniform Resource Identifier (URI): Generic syntax. <http://tools.ietf.org/html/rfc3986>.
- Caire, G. (2007). *JADE Tutorial - JADE programming for beginners*. TILAB.
- Chen, H., Finin, T., and Joshi, A. (2003). An intelligent broker for context-aware systems. In *Adjunct Proceedings of Ubicomp*.
- Coutaz, J., Dearle, A., Dupuy-Chessa, S., Kirby, G., Lachenal, C., Morrison, R., Rey, G., and Zirintsis, E. (2003). Working document on gloss ontology. Technical report, Global Smart Spaces Project IST-2000-26070.
- Fielding, R. T. (2000). *Architectural Styles and Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, UNIVERSITY OF CALIFORNIA, IRVINE.
- Gibbons, P., Karp, B., Ke, Y., Nath, S., and Seshan, S. (2003). Irisnet: an architecture for a worldwide sensor web. *Pervasive Computing, IEEE*, **2**(4), 22–33.
- Gu, T., Pung, H. K., and Zhang, D. Q. (2005). A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, **28**(1), 1 – 18.
- Huhns, M. N. and Singh, M. P. (2005). Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, **9**(1), 75–81.
- Kansal, A., Nath, S., Liu, J., and Zhao, F. (2007). SenseWeb: an infrastructure for shared sensing. *IEEE Multimedia*, **14**(4), 8–13.
- Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J., Serra, B., and Spasojevic, M. (2002). People, places, things: web presence for the real world. *Mob. Netw. Appl.*, **7**(5), 365–376.
- Ljungstrand, P., Redström, J., and Holmquist, L. E. (2000). Webstickers: using physical tokens to access, manage and share bookmarks to the web. In *DARE '00: Proceedings of DARE 2000 on Designing augmented reality environments*, pages 23–31, New York, NY, USA. ACM.

- Ostermaier, B. and Bolliger, P. (2008). Creating location-based services by utilising a web of places. In *SAINT '08: Proceedings of the 2008 International Symposium on Applications and the Internet*, pages 161 – 164. IEEE Computer Society.
- Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful web services vs. big web services: Making the right architectural decision. In *17th International World Wide Web Conference (WWW2008)*, Beijing, China.
- Román, M., Hess, C., Cerqueira, R., Campbell, R. H., and Nahrstedt, K. (2002). Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, **1**, 74–83.
- Thomas, S., Claudia, L.-P., and Korbinian, F. (2003). Cool: A context ontology language to enable contextual interoperability. In D. I. Stefani Jean-Bernard and H. Daniel, editors, *4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2003)*, volume LNCS 2893, pages 236 – 247.
- Vinoski, S. (2008). Serendipitous reuse. *Internet Computing, IEEE*, **12**(1), 84–87.
- Wang, M.-M., Cao, J.-N., Li, J., and Dasi, S. K. (2008). Middleware for wireless sensor networks: A survey. *Journal of Computer Science and Technology*, **23**(3), 305–326.
- Weiser, M. (1991). The computer for the 21st century. *Scientific American*, **265**(3), 94 – 104.
- Whitehouse, K., Zhao, F., and Liu, J. (2006). Semantic streams: A framework for composable semantic interpretation of sensor data. In K. Römer, H. Karl, and F. Mattern, editors, *EWSN*, volume 3868 of *Lecture Notes in Computer Science*, pages 5–20. Springer.
- Wilde, E. (2007). Putting things to rest. *UCB iSchool Report*.
- Wilde, E. (2008). Open location-oriented services for the web. *UCB ISchool Report*.
- Ye, J., Coyle, L., Dobson, S., and Nixon, P. (2007). Ontology-based models in pervasive computing systems. *The Knowledge Engineering Review*, **22:4**, 315–347.