Arnaud Germis

An Open Home Automation System

Analysis And Implementation of
New Solutions for HomePort

Master Thesis
Sept 2009 - May 2010

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
DK–9220 Aalborg Ø
DENMARK

Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:
An Open Home Automation System - Analysis And Implementation of New Solutions for HomePort

PROJECT PERIOD:
SSE3-SSE4,
Sept 1st 2009 -
May 25th 2010

PROJECT GROUP: d623a

AUTHOR: Arnaud Germis
agermis@gmail.com

SUPERVISOR: Ulrik Nyman

NUMBER OF COPIES: 3

REPORT PAGES: 97

APPENDIX PAGES: 13

TOTAL PAGES: 110

SYNOPSIS:

More and more devices of our daily life are computer-based. The normal development direction has turned from isolated and purpose-specific devices to collaborative and multi-purpose devices.
Many technologies were developed in parallel. Furthermore, many categories of devices have their own standards and means of communication.
The objective of this work is to provide a home-automation system. The system is based on the design of the HomePort system. It was designed to offer a distributed communicating architecture that provides an added value. It is open to new vendor's devices. It uses a layered and service-oriented architecture to fulfill its goals.
Thereafter, various solutions to problems that were left for future work in the initial HomePort architecture are described. Solutions include the automatic discovery and configuration of the components of the system.
Additionally, it includes the description of different security mechanisms, at various system levels, compatible with the rest of the system. The diverse solutions are presented, analyzed, and criticized according to the system's requirements.
Techniques to make the system redundant and scalable are described.
Finally, these new facets are implemented to be tested.

Preface

This master thesis is the final work in order to receive the Master of Science degree in Software Systems Engineering. It was written at the research unit of Distributed Systems and Semantics in the Department of Computer Science at Aalborg University, Denmark. It is based on the earlier SSE3 report, “An Open Home Automation System Focused on Automatic Configuration And Security”. However, chapters were enhanced and new one were added to compose this thesis. Additionally, the implementation was largely extended and further improved.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Ulrik Nyman, for his many useful and insightful advices and for reading my (too often) poorly spelled drafts. In addition, I am grateful for the help provided by Arne Skou to access the HomePort material and equipment needed for the implementation.

Thank you to the few people who always believed in me in every situation, my parents of course, but not only. If it were not for them, none of this would ever be possible. Concerning the others, I was lucky enough to always ignore them.

Both the Université Libre de Bruxelles and Aalborg University gave me access to an high-quality education. I acknowledge the chance that was given to me.

Finally, I would like to thank the many people who helped me during these two amazing years in Aalborg. These years were extremely rich, both, from an academic and personal perspective. Two years that changed me forever.

All this made me better prepared to start a new chapter of my life.

Thank you.

“*What is now proved, was once only imagined.*”
– William Blake

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

INTRODUCTION

MOTIVATION

More and more devices of our daily life are computer based. This trend can be seen in many different areas of home equipment. Most communication systems, including land-line phones, cellphones, and -obviously- the *Internet*, are based on electronic components that run specifically designed programs. In the entertainment area; music, movies, books, and games are available in digital formats. In parallel, devices were adapted or new ones were created to use these media. In the case of video games, a whole new industry was created based on them and gaming consoles were developed. This pattern is true as well for cars, kitchen appliances, and home heating systems that become increasingly electronic and software-based.

Overall, electronic and computer-based devices are more reliable than purely mechanical devices, based on equivalent functional-requirement complexity. Indeed, electronic components are not subject to the same physical deterioration as mechanical ones. Additionally, they offer unique and powerful possibilities. For example, it is possible to modify the behavior of a device after it was build, by changing its *firmware* or *software*. It is also easier for users to customize their devices to fit their needs. Contents of diverse kind can be copied and exchanged in an efficient and cheap manner. Finally, these devices appear smarter to the users.

The normal development direction has turned from isolated and purpose-specific devices to collaborative and multi-purpose devices. Nowadays, most cellphones cannot only make phone calls, but also surf the *Internet*, take pictures, interact with other devices (through *Bluetooth*), run user applications, etc. They interact with the cellphone network, the *Internet*, various wireless networks, and home computers. The same is true for computers, gaming consoles, and media players.

All these devices need to interact with each other through various networks. Some networks are specific for a segment of devices and others are general for heterogeneous devices. Ultimately, the *Internet* became a global network to share content and information.

This trend to make daily-life devices digital happened in a disorganized manner. Many technologies were developed in parallel. Furthermore, many categories of devices have their own standards and means of communication. In addition, devices are diverse, and they have different constraints and resources. In order to obtain truly collaborating devices, a system that can handle various technologies is needed.

Such a system would help equipment to communicate, and act in an efficient and smarter manner. Moreover, it has to be flexible enough to adapt to numerous devices and their needs. The system can help to meet the new challenges that are low-energy consumption, easy configuration, security and affordable price for such a complex system.

OBJECTIVE

The objective of this work is to provide a home-automation system. The system is based on the design of the *HomePort* system. The *HomePort* system was designed to offer a distributed communicating architecture that provides an added value. It uses a layered and service-oriented architecture.

The goal of the project is to analyze the different choices (and possible alternatives) made during the design of the *HomePort* system. It suggests solutions to the problems left for future work (security, automatic discovery, and setup) in the original work. Based on the analysis, it suggests additional features useful in the context of home automation.

It emphasizes the automatic configuration, security aspects, scalability, and redundancy. These different and related aspects are important in order to be simple and safe to use for users. Finally, it implements a working solution and tests it.

CONTRIBUTION

The current work presents the context in which home-automation systems are used. It describes and compares home-automation related technologies in general, and in the specific case where a service-oriented approach is used. Moreover, different technologies to enhance the original HomePort system are also presented.

As a starting point, the requirements of the system are defined from the existing home-automation literature. Thereafter, various solutions to problems that were left for future work in the initial HomePort architecture are described.

Solutions include the automatic discovery and configuration of the components of the system. It applies the *Devices Profile for Web Services* to the system. In the case of the event notification, it is compared with existing solutions. In order to reach acceptable user requirements, it extends and defines new functionalities, such as the group and dependency identifiers, and dynamic component discovery.

Additionally, it includes the description of different security mechanisms, at various system levels, compatible with the rest of the system. It starts by defining a security perimeter of the system. It explains the extend to which the system can protect the user, given the constraints. It proposes different possible solutions to the threats.

Moreover, these solutions are implemented to be used and integrated with the *HomePort* system. The implementation context is also described. Choices made for the implementation are described and possible alternative solutions are presented.

Finally, the performance of the implementation is measured and analyzed. It underlines limitations of this implementation on the available hardware equipment. Additionally, the consequences of the implementation based on the design are emphasized.

Overview

The current work follows a logical structure. It starts by introducing the problem and the surrounding context in the present part.

Chapter 2 on page 15 introduces the background significant for the present work. In particular, it presents theoretical concepts and definitions, and relevant technologies. Many references are used to document this part.

Subsequently, Chapter 3 on page 33 defines the requirements of the home-automation system that is presented. Specifically, it states goals, features, and constraints generally found for such a system. References and use case examples of the requirements are also presented.

Based on the requirements of Chapter 3, Chapter 4 on page 45 analyzes the design of the system. In addition, it describes and justify choices made. Many examples are integrated in order to better understand the usage scenario. Chapter 5 on page 73 presents the implementation of the designed system. Moreover, it details the working solution.

Chapter 6 on page 81 presents performance measurements of the implementation and analyzes them. They could help deciding between design alternatives.

Finally, Chapter 7 on page 89 concludes the presented work by summarizing key findings and results. It also presents future possible developments.

The Appendix contains interfaces of the system (Web browser and command-line), the mandatory thesis résumé, and a CD-Rom with the current work.

CHAPTER 2
BACKGROUND

This part provides a theoretical background about the topics used for this work. Section 2.1 on this page introduces the vast concept of home automation. Furthermore, it gives examples of existing systems in homes. Section 2.2 on page 17 presents the different technologies to connect home devices. It is subdivided in two parts. The first one is devoted to some of the wired network types. The second one is about the main wireless technologies. Section 2.3 on page 21 focuses on the protocols commonly used to connect heterogeneous devices. Different concepts about distributed systems are described in Section 2.4 on page 22. Additionally, Sections 2.5.1 and 2.5.2 on page 24 present architectural styles used to increase interoperability. Moreover, Section 2.6 on page 27 contains methods to secure electronic systems. Furthermore, Section 2.7 on page 29 reviews proposals to facilitate systems' interconnection. Finally, Section 2.7 on page 30 defines terms used in this part.

2.1 HOME AUTOMATION

In the last 15 years, the **entertainment equipment** in homes were subject to radical changes. It went from a system with mostly analogical devices and media contents to a system mostly digital[10]. Nowadays, songs, pictures, books, newspapers, and movies are available in digital formats. Television and radio shows are broadcast in digital formats as well. Often, all these media are also available over the *Internet*.

2.1.1 ENTERTAINMENT SYSTEM

Devices to enjoy these contents evolved as well. New televisions can display high-definition digital formats. Some radios use digital audio broadcasting (*DAB*)[51]. Home and portable media players are essentially all-digital[1]. Computers and video game consoles can store and use large quantities of digital formats. Cameras take pictures and movies in digital formats. [9, 10]

There is also a trend to connect digital devices to the *Internet*. Computers in developed

[1]DVD and Blu-ray players, personal video recorders (*PVR*), home audio system, portable music players,...

countries almost always access the *Internet* with broadband connections. It is also not rare to have cellphones, portable music devices, and video game consoles connected.

These types of equipment increasingly interact. In houses with multiple computers, the *Internet* connection, files, or the printer are often shared over a network. It is possible to play contents stored on a computer (movies, pictures, or songs) on a television. This is part of the phenomena of ubiquitous computing. *More and more objects of the everyday life become computing devices that can collaborate with each other. All these devices have very different characteristics, functionalities, and resources.* [9]

All these technologies are part of modern homes and change users' habits. It now seems that revolution in the entertainment home system was only the beginning of a vaster revolution in home systems.

2.1.2 BUILDING AUTOMATION SYSTEM

A building automation system (BAS) is a large concept that regroups all the systems to control buildings' environment. The first interest comes from large organization and it is mainly used in industrial buildings (offices, factories, laboratories, shopping centers,...). However, they are also used now in recent houses as part of a home automation system. [21]

2.1.2.1 Heating, ventilating, and air conditioning (HVAC) system

The heating, ventilating, and air conditioning (HVAC) system controls the climate (temperature, humidity,...) inside a building. This system is made of sensors, switches, and controllers. It is mainly used to reduce the building energy consumption[2]. *HVAC* control is increasingly done on computers. It has the advantage that the different parts of the system can coordinate. Therefore, they can be more efficient in their individual tasks. [21]

2.1.2.2 Lighting control system

The lighting control system is also part of the *BAS* family. *It is responsible of commanding when, where, and which lights are to be used.* It can include (light and motion) sensors, switches, dimmers, clocks, motorized blinds, and controllers. Such a system offers a better management and control of the different lights. Additionally, it usually proposes new functionalities. Finally, it helps controlling the energy consumption of the building. [21]

2.1.2.3 Safety and security alarm system

The safety and security alarm system is another system belonging to the *BAS* category and commonly found. *It detects alarm conditions and pass it to appropriate alerting system (local or remote alarms).* Security system can include access control devices. It can include (motion, glass break, smoke, gas,...) sensors, alarms, controllers. In case of an alarm, such a system permits to coordinate policies between the various parts of the system. [21]

[2]The interest for *HVAC* systems increased after oil price shocks of the 70's.

2.1.3 Definition

All these systems have very different constraints. They can be essential for the building to work normally and have to follow safety regulations. However, they have in common that they can increasingly be found in modern industrial and domestic buildings. Additionally, from an equipment perspective, they could share some devices (sensors, network, controllers, ...).

In the last decade, much research was made to intensify the collaboration between these subsystems. There are many examples of how the comfort, safety, or energy consumption could be improved by making them work together. The collaboration between devices increases the value of the overall system. In addition, other system can be integrated such as houseplant watering, domestic robots, management of the house's food and furniture, ... *Connecting, accessing, and controlling buildings' subsystems in order to increase their usefulness and efficiency are key goals of the home automation. It is also sometimes referred to as domotic systems.* [9, 10, 21]

2.2 Network Communication Protocols

This section introduces the most common network technologies that are used in houses. Its goal is to present ways to connect the home subsystems described in Section 2.1 on page 15. The first subsection focuses on wired networks. The second one presents the main wireless systems. At the end of the section, Table 2.1 on page 20 summarizes the technologies.

Traditionally, networking technologies are described following an abstract model called the **open system interconnection reference model** (*OSI* Reference Model or *OSI* Model). This model contains 7 layers to describe a network architecture. From the highest to the lowest, these layers are: the application, presentation, session, transport, network, data-link, and physical layer. [18]

This section only focuses on technologies on the physical layer which describes electrical and physical specifications between devices, and the local area network (*LAN*) data-link layer which describes means to transfer data.

2.2.1 Wired Networks

Wired networking technologies offer a high bandwidth and a low interference rate. Additionally, they can supply power to connected devices.

Ethernet (IEEE 802.3)

Ethernet is a standardized technology (IEEE 802.3) used from around 1980 to nowadays. It is the most common type of local area network (LAN). In addition, it works with twisted copper wires or optical fibers. The maximum speed depends on the cable's category. All the cables of the devices can be connected together trough *Ethernet* hubs or switches. [44]

Universal Serial Bus (USB)

Universal Serial Bus (USB) is the leading technology[3] *to connect devices to computers.* A *USB* network is made of one host with, at least, one controller and one device. Controllers have ports (or plugs) to which the devices connect with a specific cable. *USB* hubs can be connected to a port to extend the total number of ports. Therefore, they form a tree structure with maximum 5 levels. A maximum of 127 devices (including the hub(s)) can be connected to a controller. [45]

USB uses specified class codes to recognize the functionality of the devices. If a device has more than one functionality (called a *compound device*), each of them is represented as a separate *USB* device with its own class code. The class code enables the controller to load drivers based on it. Furthermore, generic drivers can be written for a class. [45]

> Example:
>
> A portable audio player can belong to the *USB* audio class but also to the mass storage class to access its memory. When the player is connected, its host automatically recognizes its two functionalities based on the two classes.

Networking Over Electrical Wires

The electrical wire network already exists in every house. The main disadvantage is that the electrical power standards are different in many countries. Therefore, there is no universal standard for this technology. The two main coexistent (but incompatible) standards are *HomePlug* and *UPA Digital Home*. [2, 7]

2.2.2 Wireless Networks

Wireless networking technologies benefits from the lack of cable installation cost. Moreover, they are very useful in places where no wired network exists, and where it would be cumbersome or impractical to use cables. Therefore, they fit well with needs of mobile devices, in particular. In contrast, they are usually subject to shorter range and lower bandwidth than wired counterparts. Additionally, they do not provide power supply. Wireless networks can work with a centralized access point that relays all the traffic, or in a decentralized ad-hoc way.

Wi-Fi (IEEE 802.11)

Wi-Fi is a device certification delivered by the Wi-Fi Alliance[3] based on the IEEE 802.11 standard. It was conceived as a "wireless Ethernet" solution. The standard contains many different wireless related technologies that evolves over time. Therefore, it is subdivided in different amendments that are referenced by a letter. When new wireless technologies appear, a new amendment is defined to provide increased range and speed by using new technologies. In this sense, amendments represent evolutions of the standard. The most

[3]There are 6 around billion *USB* devices and 2 billions are sold a year. [30]

common amendments in home network are *802.11b, 802.11g*, and *802.11n*. The *802.11b* standard is the oldest and *802.11n* is the most recent. Additionally, some amendments of the standard define how to secure a wireless network using cryptography. [39, 3]

BLUETOOTH

Bluetooth is a specification to transfer data over a short distance (100 m) between resource-limited devices. It is used by cellphones, remote controller (mouse, keyboard, remote control, ...), GPS[4], and other highly portable devices. [38]

WIRELESS USB

Wireless USB is a specification for low-range high-bandwidth transfer of data. It was conceived as a "cable-less USB" solution. In this sense, it offers similar functionalities as *USB*. The maximum bandwidth depends on the distance between the communicating devices. Up to three meters, the maximum bandwidth is 480 *Mbit*/*s*. Between three and ten meters, it is 110 *Mbit*/*s*. [45]

ZIGBEE

ZigBee is a specification[5] *similar to Bluetooth, but is intended to be simpler, cheaper, less power consuming, and more secure.* In contrast, it offers a lower range and bandwidth than *Bluetooth*.

A *ZigBee* network contains three different kind of devices: *ZigBee* coordinator (*ZC*), *ZigBee Router* (*ZR*), and *ZigBee End Device* (*ZED*). The *ZC* is the root of the network tree. It stores the network and security related data. The *ZR* is in charge of relaying data from and to other devices (*ZC, ZR*, and *ZED*). *ZEDs* only communicate with the *ZC* or *ZRs*. It does not relay data. A *ZED* is designed to be cheap to manufacture. [5]

Z-WAVE

Z-Wave is a proprietary specification[6] *designed to control devices in residential and commercial buildings.* It focuses on aspects such as low-power consumption, low-protocol overhead, cheap manufacturing cost, and decreased interferences with the radio frequency network. [4]

SUMMARY

Table 2.1 on the following page presents a summary of all the home network technologies mentioned above. It provides names, network types, the medium used, transmission speeds, and maximum distances.

[4]The Global Positioning System (GPS) is global navigation satellite system.
[5]The specification is only freely available for non-commercial use.
[6]The specification is only available under a non-disclosure agreement.

Name	**Type**	**Medium**	**Speed**	**Distance (max.)**
Ethernet	LAN	Twisted pair	10 Mbit/s to 1 Gbit/s	100 m
USB	PAN	Specific twisted pair	1,5, 12, or 480 Mbit/s	5 m
HomePlug	LAN	Electrical wiring	14 Mbit/s to 200 Mbit/s	200 m
UPA Digital Home	LAN	Electrical wiring	200 Mbit/s	200 m
802.11b	WLAN	Radio frequency	11 Mbit/s max.	(indoor) 45 m
802.11g	WLAN	Radio frequency	54 Mbit/s max.	(indoor) 45 m
802.11n	WLAN	Radio frequency	600 Mbit/s max.	(indoor) 90 m
Bluetooth	PAN	Radio frequency	1 Mbit/s to 10 Mbit/s	100 m
Wireless USB	PAN	Radio frequency	110 Mbit/s to 480 Mbit/s	10 m
ZigBee	PAN	Radio frequency	20 Kbit/s to 250 Kbit/s	(outdoor) 75 m
Z-Wave	PAN	Radio frequency	9,6 kbit/s to 40 kbit/s	(outdoor) 30 m

Table 2.1: Summary of the home network technologies.

2.3 High-Level Communication Protocols

This section presents communication protocols from the network layer to the application layer. Section 2.3.1 on the current page exposes protocols commonly used on the *Internet* (known as the *Internet standards*). However, the same protocols are used in all kind of networks. Section 2.3.2 on the following page introduces a protocol to automatically discover and configure devices on a network.

2.3.1 The Internet Standards

The following part describes various standards used over the *Internet*. These standards help the communication between devices (and their users, at the end). They are presented from the lowest level of abstraction to the highest. Additionally, they belong to different architectural *OSI* layer. Therefore, they build upon each other to work.

2.3.1.1 Internet Protocol (IP)

The Internet Protocol is the most common protocol of the network layer in the OSI reference model (see Section 2.2 on page 17 for more details). *It is the protocol at the heart of the Internet. Most devices support it.*

The protocol is connection-less. The data are fragmented in units called packets for efficiency reasons. A packet is made of a header to route the packet over the network and the data. The header contains the source and destination *IP* addresses, the source and destination ports, and other options. *IP* does not provide guaranteed delivery, duplicate avoidance, integrity, or correct ordering of packets. These properties, if needed, are the responsibility of higher protocol layers (usually the transport layer with the **Transmission Control Protocol** (*TCP*)). Therefore, it belongs to the best effort of delivery protocols' category. The **User Datagram Protocol** (*UDP*) is a very light protocol that offer the same properties as *IP*.

Additionally, *UDP* gives the possibility to send a packet to a set of devices (**multicast**), or to every device (**broadcast**) in the network. These are efficient techniques when the same packet has to be sent to many devices.

Currently, two versions of the *IP* are used: the fourth (**IPv4**) which is the most common and sixth (**IPv6**) which offers more *IP* addresses besides other functionalities. *IPv6* does not implement the broadcast. However, a similar effect can be achieved by using multicast alone.

2.3.1.2 Hypertext Transfer Protocol (HTTP)

The ***hypertext transfer Protocol*** *(**HTTP**) belongs to the application layer in the OSI reference model* (see Section 2.2 on page 17 for more details). *It is a distributed state-less protocol used to transfer hypermedia resources. It uses a request/response architecture.* [40]

The client sends a request with a **header** and optionally a **body** to a server. A header contains a **request method**, an *URI*, the version of the protocol, optionally client information. A body is the representation of a resource transfered.

The server replies with a **status line** and optionally a body. The status line contains the version of the protocol, a code describing the response's type (success or type of errors), and other response's meta-information.

There are four **request** methods to manage a resource: GET, POST, PUT, and DELETE. The GET method returns a representation of the requested resource. The POST method submits data to be processed by the requested resource. The PUT method uploads a representation of the requested resource. Finally, the DELETE method deletes the requested resource. The GET, PUT, and DELETE methods are idempotent. It means that multiple identical requests do not change the resource. Additionally, the GET method never modifies the resource on the server.

2.3.1.3 Extensible Markup Language (XML)

***Extensible Markup Language** (**XML**) is a set of specifications to inter-operate, store, and transfer data between systems. It is designed to be simple, generic, structured, human and machine readable, and used over the Internet.* It uses a textual data format with *Unicode* encoding[7].

XML documents are structured with **elements**. Elements are defined by tags. There are two ways to define an element: with starting and ending tags that go by pair, or with empty-element tag that does not contain any other element. The document's hierarchical structure comes from the fact that other elements can be defined between an element's starting and ending tag. Therefore, there is a multi-level hierarchy with elements (parents) containing other elements (children). An element can also contain **attributes**. An attribute is a pair (inside a starting or empty-element tag) defining a unique name and an associated value.

An *XML* schema is a document, written in some kind of definition language, that specifies the abstract content of one or more *XML* documents. It defines the mandatory and optional elements and attributes. Moreover, it specifies the data interpretations.

Many other serialization formats exist, such *JSON*. *JSON* (for *JavaScript Object Notation*) is an open, language-independent, human-readable, data interchange standard. It is derived from *JavaScript* and it is an alternative to *XML*.

2.3.2 Universal Plug and Play (UPnP)

There are different standards to facilitate interactions between certain devices. One of them is *Universal Plug and Play* (*UPnP*). It is a set of protocols developed by the *Universal Plug and Play Forum* (*UPnP Forum*) and built upon Internet standards [34]. *UPnP can automatically and seamlessly discover and share devices that support the standard.* The problem is that UPnP relies heavily on Internet standards (*IP, TCP/UDP, HTTP, XML,...*). Resource limited devices lack the capacity to handle these resource demanding protocols. [36]

2.4 Distributed Systems

A distributed system consists of various, independent, possibly heterogeneous, computing nodes. These nodes collaborate together in order to achieve a goal (delivering content, processing data, ...). They need to exchange data to achieve their goal. Just like regular program run on a single computing node, they need input and output data, but additionally they might need data to exchange their current state or synchronize with each other. This is done by *message passing*.

[7]Unicode is an international character encoding standard supporting many writing symbols.

The nodes communicates according to a communication protocol over a medium (network) (see Section 2.3 on page 21 for more details) to send messages. The network is subject to *latency* and is limited to a maximum *bandwidth*. Additionally, it can suffer of *failures* and *transmission errors*. All these issues are due to the physical limitations of equipment.

These network problems are naturally passed to communication protocols. Protocols offer different services to solve or mitigate these limitations. These solutions include retransmission of messages, message integrity checking, network redundancy, message ordering, etc. *Protocols offer a level of abstraction to the physical limitations of passing messages between devices.* [8]

Additionally, nodes are also subject to *failures, memory corruptions, and computational errors*. However, one of the advantages of distributing the computation of a problem on different nodes is that it is possible to rely on the remaining nodes if some nodes fail. Therefore, *the system can tolerate some failures* of individual nodes. See Section 2.4.1 on this page for more details.

Another possible advantage is that *the system can scale* to the problem by integrating new nodes in the system. Indeed, computation can be done in parallel on different nodes and not an a single node. It enables the federation of resources (computational, memory, storage, ...). However, there is an overhead to the use of distributed system that is due to the message passing between nodes. See Section 2.4.2 on the current page for more details.

2.4.1 Replication

In order for a distributed system *to increase its reliability and to be fault-tolerant, it needs to replicate data on different nodes*. Indeed, if a piece of data is present on only one node and the node fails or is unreachable, it is unavailable to the system. Therefore, the system needs redundancy of its data by replication in order to offer reliability.

Another advantage of replicating data is *the possibility to balance the load on different nodes*. Indeed, all nodes storing a piece of data can distribute it to the rest of the system. That means that the load serving or processing data can be balanced on all the nodes where the data is available.

Different strategies exist to replicate data. It usually involves that when a node changes the state of the distributed system, it broadcasts the change to a set of other nodes. The other nodes acknowledge the change and the new state is then committed. When a node fails, the remaining nodes agree on a state that is represented by a set of values. The algorithm to agree on these values is called a consensus algorithm (such as the famous Paxos algorithm[22]).

This replication process can become inefficient because every time a change is made it needs to be broadcast and acknowledged by a set of device. It generates communication overhead and latency. One solution is to delay and group changes. However, the data is at risk as long as it is not committed in the system. Another solution is to reduce the number of replicates. However, it increases the chance that all the copies are unavailable.

2.4.2 Scalability

The scalability is the capacity for a distributed system to handle a growing load by adding new resources. A usual requirement for an optimal scalability is that the resources added grow linearly (and not exponentially) to the capacity to handle load in the system. It means that the time to solve a fixed problem is linear in the number of resources available in the system. However, this is not always possible as not all operations can be executed in parallel.

2.5 Web Service (WS)

Web services (WS) are applications that are executed on a server to fulfill client requests. Transactions are transmitted between clients and servers through a network, such as the *Internet*.

The distributed nature of *WS* make then unreliable and subject to concurrent access, partial failure, and latency issues. Therefore, the exception handling is an essential part of a *WS* design. [47, 20]

WS are often described with the *web services description language* (*WSDL*). It is an *XML*-based language that gives means to describe the different components of a *WS*: the type; endpoint (typically a *URI* to contact the *WS*); operations (the input and output of a defined transaction); interface (set of operations); and the binding (the protocol used for the interface). [50, 20]

It typically uses *HTTP* (see Subsection 2.3.1.2 on page 21 for more details) to convey transactions, *XML* (see Subsection 2.3.1.3 on page 22 for more details) to serialize them, and other *Internet* standards for interoperability.

WS often use service-oriented and *REST* architectures as described in Sections 2.5.1 and 2.5.2 on this page. Both architectural styles fit the usual *WS* requirements. However, they are optional and they can be used individually or jointly. Ways to discover *WS* are presented in Section 2.5.4 on page 26. [20]

2.5.1 Service-Oriented Architecture (SOA)

A service-oriented architecture is a set of design concepts aimed at delivering software functionalities independently. A functionality is a defined service which does only one (possibly elaborate) action.

This architectural style enables the collaboration between remote and heterogeneous software. The software presents an abstract interface that hides the underlying complexity and structure to the clients. Additionally, it follows different principles: modularity, standard compliance, and functional autonomy.

2.5.2 Representational State Transfer (REST)

*The **representational state transfer (REST)** is an architectural style for request/response distributed protocols.* It was developed in parallel with the *HTTP* which is the most wide spread example of such a protocol. Moreover, it ensures desirable properties for a service-oriented architecture. Therefore, the two concepts are complementary. [16]

Its goals include the generalness of interface, the device autonomy, the possibility to seamlessly add intermediate devices[8], and the scalability to new and additional devices. To achieve these goals, a *RESTful* protocol follows these principles [16]:

Client-Server It separates the concern of the client and the server. On one hand, the client is responsible for the user representation of the data which improves the portability of

[8]Intermediate device can be added to enhance the response time or security without interfering with the existing devices, for example.

the user interface. On the other hand, the server is responsible for the storage of the data which increases the server simplicity and scalability. Overall, it allows clients and servers to evolve independently.

Stateless The protocol is stateless. A request contains all the necessary information to be processed by the server. The server does not store any context. It increases the device autonomy and the overall scalability.

Cache A response specifies if it can be cached. A cached response can be reuse by the client or an intermediate device to respond to an equivalent request. It increases the scalability and adds the possibility of having intermediate cache devices.

Uniform Interface A generic uniform interface is presented to all the protocol components. Therefore, they can evolve independently and are always aware of protocol transactions. This principle generates four constraints: identifiable resources (with a *URI* as an example) (see Subsection 2.3.1.2 on page 21 for more details), manipulation of resources through representations [9], self-descriptive transactions, and hypermedia as the engine of application state (related external resources are identified).

Layered System A layered-client-server (also known as multi-tiered) architecture is used. A transaction goes seamlessly through multiple layers. The possible intermediaries are invisible to clients and servers. The layering reduces the overall coupling between layers and enables intermediate-based load balancing, monitoring, or security checking.

Code-On-Demand (Optional) The client's functionalities can be extended by providing codes or scripts to be run on the client. It reduces the client's complexity and enables evolvements.

Example:

Many web servers provide *JavaScript* code in their *HTML* pages that are executed by the client's web browser. It makes web pages more dynamic with customized scripts without additional burden of the web server. The client simply need a generic *JavaScript* interpreter instead of having numerous complex functionalities.

2.5.3 SIMPLE OBJECT ACCESS PROTOCOL (SOAP)

In general, *SOAP serves mainly as a tool for message exchanges and remote requests* (Remote Procedure Calls (RPC)). It was developed by two leading IT companies (IBM and Microsoft). *Originally the developers of SOAP intended to create it for more purposes [49], two of them are:*

1. *Provide a standard object invocation protocol built on Internet standards, using HTTP as the transport and XML for data encoding.*

2. *Create an extensible protocol and payload format that can evolve.*

[9]The request information is enough. No state is needed on the server.

MESSAGE FORMAT

SOAP messages use the *XML* syntax which makes them readable for humans and easily processable by applications. Indeed, there are a lot of reliable *XML* handling utilities (*parsers, query processors, validators*, ...). The message it-self is composed of a header and a payload where the former is optional and might provide useful information for the *SOAP* engine which processes it. The latter is compulsory and contains the data for the target *SOAP* engine.

ANALYSIS

SOAP can be used as a reliable communication protocol between web services due to the fact that it works perfectly on existing Internet infrastructures. Another advantage is that *SOAP* is platform independent. It can run everywhere as long as it is possible to communicate using the *HTTP*. Often, *SOAP* is criticized for its verbosity which is caused by the *XML* message syntax. It is slower in comparison with other remote process call mechanisms, but the speed was not the prime factor when *SOAP* was designed. It offers more flexibility for the data encapsulation and the remote procedure call than *REST*. Indeed, web-service designers can define new and arbitrary elements and request methods. However, by doing so, it might decrease the interoperability and not take advantage of existing mechanisms.

2.5.4 SERVICE DISCOVERY

Before using a web service, the client has to know the existence of the service. *WSDL* (see Section 2.5 on page 24 for more details) describes a *WS* and the possible interactions. However, the *WSDL* document has to be available to clients. *The goal of the service discovery is to find WS*. Obviously, it is possible to manually encode the available services during the client development. However, if new *WS* appear, the client does not benefit from them. [47]

There are three approaches to automatically discover *WS*:

The registry A registry centrally stores the *WS* descriptions. The registry controls the published *WS*.

The index An index points to external *WS* descriptions. It has no control over the description, but controlled the kind of descriptions indexed.

Peer-to-Peer (P2P) discovery At discovery time, the client requests the *WS* descriptions know by its neighbors. The descriptions are not centralized.

These different approaches can be combined.

2.6 SECURITY

This section introduces security aspects and some protocol solutions. General security concepts are briefly presented in the following Subsections. The following parts describe various security protocols acting at different network layers.

2.6.1 SECURITY CONCEPTS

Information security regroups many different concepts. There exist many different models to describe security requirements. The most usual requirements to ensure a transaction's security (known as the *CIA* triad) are: [6]

- Confidentiality: the transaction content is only disclosed to authorized parties.
- Integrity: the transaction is only emitted and modified by authorized parties.
- Availability: the system is available at all times.

To ensure the two first requirements different techniques were developed. The availability requirement mandates more abstract constraints such as the data and service redundancy.

CRYPTOGRAPHY

In cryptography, the encryption process transforms the information into seemingly meaningless data by using an algorithm and a key. The encrypted data can only be decrypted with the key or by the extremely long process of testing all the possible keys. There are two kinds of encryption algorithms: symmetric-key encryption and public-key encryption. [6]

Symmetric-Key Encryption

In the case of symmetric-key encryption, the same key is used to encrypt and decrypt data. It is possible to use this method to ensure that the data are from a party knowing the secret key. Additionally, the identity of a party can be assessed by parties sharing a common secret key. [6]

Example:

In computer systems, a specific example is when a user identifies it-self (authenticate) with one of the many challenge/response protocols. The client sends an authentication request to the server. The server replies with a challenge to the client. The challenge is made in such a way that, only a client knowing the shared-key can respond. A secret password is a specific kind of key shared between the user that knows it and the system that stores it.

Public-Key Encryption

In the case of public-key encryption, a pair of keys is used. One key is public and is used to encrypt data. The other key is private and is used to decrypt the public-key encrypted data. This method can also be used to sign data. Signing data ensure to anybody having the public key that the data are from the party possessing the private key. Additionally, the identity of a party can be assessed by proving that it can decrypt with its private key data encrypted with its public key. [6]

2.6.2 Authentication

There are different ways to authenticate a party. A common secret key can be exchanged, as in the case of password or passphrase. Nowadays, it is the most common way to authenticate a user. The main drawback is that the key has to have a high entropy and being long enough. Otherwise, the key can be potentially guessed by trying all the possible keys. The password authentication is supported by *HTTP*, but the sent password is not encrypted. Therefore, this system provides very little security. [6]

Another way is to use public-key signatures. *A model of trust has to be established with this technique. The two most common are the web of trust and the certificate authority models. In the* ***certificate authority*** *model, the identity of the parties is assured by a trusted third party that signs the public keys.* Therefore, all parties can trust each other because the public keys, that they use to sign transactions, are them-self signed by a common trusted third party.

In such a system, trusted third-party public keys are stored in the device. One advantage is that devices do not need prior knowledge of the other devices to authenticate them. Another advantage is that the trust into a party can be revoked by the trusted third party. [6]

2.6.3 IPsec

IPsec is a protocol (on top of IP) to ensure the confidentiality and integrity of IP packets (see Subsection 2.3.1.1 on page 21 for more details). *IPv4* has an optional support of it. In contrast, *IPv6* requires the possibility to use *IPsec*. The packet content is encrypted and verified by using a symmetric-key encryption algorithm. To the exception of the content being encrypted, an *IPsec* packet is equivalent to regular *IP* packet. [42]

2.6.4 Hypertext Transfer Protocol Secure

Hypertext Transfer Protocol Secure *(**HTTPS**) is a protocol that provides, additionally to the HTTP functionalities* (see 2.3.1.2 for more details)*, confidentiality and integrity of the transactions* [41]. This is done by using the transport layer security (*TLS*) that ensures the security of the transport layer [43].

The protocol can authenticate both the client and the server (mutual authentication) or only one party.

In the server-authenticated mode, the client can still authenticate it-self by another method such a *login/password* authentication. All data exchanged are encrypted.

HTTPS is commonly used to secure transactions on the Internet. Therefore, it is widely supported by web browsers and network software libraries. Being a different protocol than *HTTP*, it typically uses a different port (*443*).

2.6.5 WS-Security

WS-Security is a protocol to ensure the confidentiality and integrity of WS transactions. It has the advantage to ensure end-to-end security between the client and the *WS* application. Intermediaries have no access to the exchanged data and caching is still possible at a lower level. Additionally, there exists standards to sign and encrypt *XML* documents [46]. [47]

2.6.6 Firewall

A firewall is an intermediate network device that filters devices' communication for security reasons. Commonly, it is used to separate the home network from the *Internet*.

The transaction filtering is based on rules that define authorized communications. Often, the firewall acts at the *IP* level. Rules defines (source and destination) *IP* addresses and ports that are authorized. See Subsection 2.3.1.1 on page 21 for more details. It can also work at higher levels by analyzing exchanged transactions.

2.7 Interconnection Between Heterogeneous Subsystems

HomePort

The ***HomePort*** *system's goal* (defined in [13]) *is to connect different subsystems in houses through an open architecture. These subsystems use different (and incompatible) protocols to communicate with their devices.*

HomePort has a service-oriented architecture (see Section 2.5.1 on page 24 for more details), and uses defined protocols and interfaces. In addition, it has the following architectural requirements:

- Modifiability: the addition of new devices or subsystems does not affect the already existing components.
- Usability: an easy to use and configure system helps to reduce the acquisition cost.[10]
- Scalability: the system supports an significant number of devices spread over a large building without affecting the performance.[11]

Furthermore, different business considerations are taken into account. The system supports partially closed protocols and it does not affect existing and future manufacturer's products.

The *HomePort* system has a four-layer architecture defined as followed:

[10]The goal is to lower the amount of time needed to configure the system by using automatic setup.
[11]Performance measurements can be conducted on a system prototype by adding new devices.

1. The **device layer** is responsible for communicating with a specific subsystem's devices. It is present in every **device**.

2. The **bridging layer** is responsible for making the subsystem's devices accessible over the *IP* network. This layer is divided into two sub-layers. One that is subsystem specific and another that is bridging to the *IP* network using the *heterogeneous network protocol* (*h-net*). The *common network adaptor interface* (*CNAI*) joins the two sub-layers, and defines how to connect arbitrary subsystems to the *IP* network. This layer is furnished by an equipment called **bridge**.

3. The **service layer** is responsible for presenting in a subsystem independent manner the devices' functionalities, and controlling the access to the devices. To fulfill its requirement, the system uses a *REST* architecture based on the *HTTP* (see Subsection 2.3.1.2 and Section 2.5.2 on pages 21–24 for more details). The access policy requirements are not defined in the current system and it is left for future work. Physically, the service layer is present in a piece of equipment called **gateway**.

4. The **composition layer** is responsible for combining heterogeneous devices. The *HomePort control logic language* (*HCLL*) defines these combinations. The *HCLL* explains the actions to take on a device based on states and events in the system. A piece of equipment, called the **controller**, is responsible for delivering this layer.

Some architectural layers can be omitted or merged for economical reasons. The process by which equipment discover each other is not defined in the current system and it is left for future work. Moreover, the mobility of the devices has not been explored. Therefore, the intermittent availability of devices is not handled.

Glossary

Authentication An authentication is the action of proving its identity to another party.. 27, 28

Client A client is a party that requests a service over a network.. 21, 24, 26, 28, 29

Device A device is an electronic component with hardware, and software or firmware.. 15, 17

Distributed system A distributed system is made of independent devices that communicate over a network in order to achieve a goal.. 24, 29

Exception An exception is an event that is not part of the normal flow of a program.. 24

Interface An interface defines how a program can request a service.. 24, 29

IP address An IP address identifies on an IP network a device with a unique address.. 21, 29

Local Area Network (LAN) A local area network (LAN) is a network that covers a reduced area such as a house or an office.. 17, 20

Packet A packet is a block of data transmitted over a network.. 21, 28

Party A party is an entity that interact with a distributed system (e.g. a client or a server).. 27, 28

Personal Area Network (PAN) A personal area network (PAN) is a small network that covers the area around a user.. 20

Port A port identifies on an IP network a device's process with a device unique number.. 21, 28, 29

Protocol A protocol is a decription of a set of actions to communicate data between parties.. 17, 19, 21, 22, 24, 27, 29

Request A request is the communication sent by a client to obtain a service from the server.. 21, 24, 26

Response A response is the reply sent by the server to a request.. 21, 24

Serialization A serialization is the action to transform conceptual data into a form that can be transmitted through a network.. 24

Server A server is a party that provides a distributed service on request over a network.. 21, 24, 28

Transaction A transaction is a request or response.. 24, 27–29

Uniform Resource Identifier (URI) A uniform resource identifier (URI) identifies a resource on the Internet with a structured string of characters.. 21, 24

Wireless LAN (WLAN) A wireless LAN (WLAN) is a more specific LAN category that does not require cables. It usually uses radio frequency technologies.. 20

CHAPTER 3

REQUIREMENTS

This part defines the multitude of requirements needed for a home-automation system. The system has different stakeholders that share different requirements. These stakeholders are the users, the device vendors, the subsystem architects, and system architects.

3.1 GOALS

The goal of the system is to connect different subsystems in houses through an open architecture. These subsystems use different protocols to communicate with their devices. The purpose of connecting these, otherwise independent, subsystems is to achieve:

- increased usefulness of the devices by combining them
- simplified use
- optimized energy usage
- advanced security

3.2 FEATURES

This section defines a list of desirable features for a home-automation system. Often, they are abstract and difficult to quantify. Each feature is described and categorized. Features belongs to different categories of responsibility that can only be achieved by mutual collaboration of the stakeholders. Additionally, motivations, references and examples are provided, whenever possible.

3.2.1 Openness and Interoperability

Vendor Independent

Description:	The system is independent of any vendor (not vendor specific).
Category:	System design (abstract)
Motivation:	The value of the system increases if it is accessible to all vendors. [24, 13]
Example:	New vendors can decide to use the system for its products.

Language Independent

Description:	The implementation of the system is independent of any programming language.
Category:	System design (concrete)
Motivation:	Limit constraints for the vendors. [24, 13]
Example:	Vendors can decide to write their implementation with the programing language of their choice.

Subsystem Independence

Description:	Subsystems work by them-self (no subsystem is needed for the others to work). This requirements has to be fulfill by vendors.
Category:	Vendor design (concrete)
Motivation:	Increase users' freedom to chose the best subsystem for their needs. [25, 13]
Example:	Every subsystem should work even if the system is made of one subsystem.

Legacy Devices

Description:	The system has to be able to work with already existing equipment and devices.
Category:	System design, vendor device
Motivation:	Reduce the total cost of the system. [20, 13]
Example:	Bridges can exist to connect legacy devices with the system.

Defined Abstract Interface

Description:	The system has defined generic interfaces available to all vendors.
Category:	System design (abstract)
Motivation:	Interoperability. [13]
Example:	-

Programmatic Control

Description:	The subsystem available functionalities has to be programmable by other subsystems.
Category:	System design (concrete), vendor design (concrete)
Motivation:	Usability. [24]
Example:	A program on a computer could (following a protocol) control the available functions of a subsystem.

Graphical Interface Independence

Description:	The system does not rely on a specific graphical user interface.
Category:	System design (concrete)
Motivation:	Users' freedom. [28]
Example:	Vendors can propose their own centralized graphical interface. Therefore, the user is free to use one or more graphical interface to manage the system.

3.2.2 Communication

Physical-Layer Independence

Description:	The system should be independent of the underlaying physical protocol.
Category:	System design (concrete)
Motivation:	Ensure the system generality.
Example:	-

Subsystem Addressing

Description:	Each subsystem should be individually addressable. Additionally, a subsystem has to offer the possibility to communicate with devices that it controls.
Category:	System design (concrete for subsystem), vendor design (concrete for devices)
Motivation:	Usability. [19, 20]
Example:	-

Discovery

Description:	A subsystem should be able to publicize its existence and discover the other subsystems.
Category:	System design (concrete)
Motivation:	Simplicity for users. [19, 20]
Example:	When a new subsystem is connected, it automatically informs of its own existence and look for other subsystems.

Description

Description:	A subsystem should be able to describe information useful for the system.
Category:	System design (abstract)
Motivation:	Interoperability. [19, 20]
Example:	When a new subsystem is connected, it sends its version, devices, dependency information, group identifiers, etc

Control

Description:	It should be possible to control a subsystem. Afterwards, it should be possible to asses the new state of the subsystem.
Category:	System design (concrete)
Motivation:	Interoperability and usability. [19, 20]
Example:	After the command to turn on a light is sent, it is possible to check if the light is on.

Eventing

Description:	It should be possible for other subsystems to know when some selected events happen.
Category:	System design (concrete)
Motivation:	Usability. [19, 20]
Example:	When the alarm system detects an intrusion, the event is sent to lightning and communication systems.

3.2.3 Scalability, flexibility, and dynamism

New devices

Description:	It should be easy to add new devices to the system.
Category:	System design (abstract)
Motivation:	Simplicity and interoperability. [26]
Example:	-

New Vendor Subsystems

Description:	The system has to be generic enough to enable the addition of new subsystems.
Category:	System design (abstract)
Motivation:	Reduce constraints for vendors. [26, 12, 13]
Example:	-

New Technologies

Description:	The system has to be able to support new technologies by adding new devices and/or subsystem software update.
Category:	System design (abstract ability), vendor devices (concrete updates)
Motivation:	Interoperability. [12]
Example:	-

Automatic Configuration

Description:	The system should use automatic setup as much as possible. In particular, user interactions to configure the system should be minimal.
Category:	System design (abstract), vendor design
Motivation:	Simplicity. [19]
Example:	-

Dependency information

Description:	The subsystem should define available functionalities to the rest of the system.
Category:	System design (concrete)
Motivation:	Interoperability. [28]
Example:	The subsystem publishes all its available functionalities.

Group Identifiers

Description:	Devices can be associated with different identifiers. This enables devices to be addressed based on their groups instead of individually.
Category:	System design (concrete)
Motivation:	Ensure scalability with large scale systems and interoperability.
Example:	For example, a device could belong to the group: “Kitchen” and “Lights”.

Inheritance

Description:	It should be possible to refine dependency information and group identifiers.
Category:	System design (concrete)
Motivation:	Ensure scalability with large scale systems and interoperability.
Example:	For example, the group identifier “Kitchen” could inherit from the group identifier “Room”.

3.2.4 Usability

Simplicity

Description:	The system should be simple from a user perspective.
Category:	System design (abstract)
Motivation:	Usability for users. [32, 13]
Example:	-

Seamless Interactions

Description:	The interaction process between the subsystems should be seamless.
Category:	System design (abstract)
Motivation:	Simplicity. [32, 13]
Example:	The user does not need to understand the interactions between subsystems.

Subsystem binding

Description:	Once authorized (for security requirements), a new subsystem should automatically bind to the system to exchange information needed to work.
Category:	System design (concrete)
Motivation:	Simplicity
Example:	-

Remote Control

Description:	The user should be able to control the system remotely, in particular over the Internet.
Category:	System design (concrete)
Motivation:	Usability and increase the value of the system. [37]
Example:	A user could turn on the heating system from the Internet.

Browser Control

Description:	It should be possible to control the system with a browser. It offers a common, standardized, and widely supported way to access information. Therefore, the system has at least this user interface. The user remains to use the interface of his/her choice.
Category:	System design (concrete), vendor devices
Motivation:	Simplicity, interoperability, and usability. [24, 37]
Example:	-

External Source Of Information

Description:	The system should be able to access available resources external to the network.
Category:	System design (concrete ability), vendor design (abstract use)
Motivation:	Usability and increase the value of the system. [37]
Example:	The system could access weather forecast or natural disaster alerts on the Internet.

Reliable

Description:	The system's reliability should be maximized by using opportunistic redundancy.
Category:	System design (abstract)
Motivation:	Improve users' experience and confidence.
Example:	If two subsystems offer the same functionality and one fails, the other should take over.

Acceptable Delays

Description:	In order to have a satisfiable user experience, communications between devices should be efficient to increase the speed.
Category:	System design (abstract)
Motivation:	Improve users' experience.
Example:	A user should not be able to notice the transmission delay between subsystems (to turn on a light for example).

3.2.5 Security

Prevent undesired actions

Local

Description:	The system should be able to prevent actions that are undesired for a subsystems devices.
Category:	System design (concrete), vendor design (rules abstract)
Motivation:	Safety. [52, 11]
Example:	For example, turning on and off lights too rapidly.

Global

Description:	The system should be able to avoid undesired actions due to subsystem interactions.
Category:	System design (abstract)
Motivation:	Safety. [52, 11]
Example:	If all subsystems turn on all equipment at the same time, it can generate problem in the electrical consumption of the house.

Environmental

Description:	The system should take into consideration the external environment.
Category:	System design (concrete), vendor design (implementation)
Motivation:	Safety. [52, 11]
Example:	Forcing the system to shut the windows when there is a pollution outside the house.

Unavoidable Issue Detection

Description:	If an issue cannot be automatically fixed, the system should warn the user.
Category:	System design (concrete ability), vendor design (detection)
Motivation:	Users' experience. [11]
Example:	For example, a subsystem has a defect.

Confidentiality

Description:	All data exchanged in the system should be considered confidential. The system should treat them accordingly.
Category:	System design (abstract)
Motivation:	Safety. [11]
Example:	Data exchanged between subsystems should be encrypted.

Authenticity

Description:	A subsystem should be able to asses the authenticity and origin of data.
Category:	System design (concrete verification), vendor design (trust)
Motivation:	Safety. [11]
Example:	The security system should be sure of the origin of a transaction.

AUTHORIZATION

Description:	Only authorized subsystems should be part of the system.
Category:	System design (concrete)
Motivation:	Safety. [11]
Example:	It is impossible for an intruder to add a subsystem to the system.

INTEGRITY

Description:	A subsystem should be able to asses that data are not tampered.
Category:	System design (concrete)
Motivation:	Safety. [11]
Example:	-

3.3 CONSTRAINTS

3.3.1 HARDWARE

The constraints defined hereafter come from hardware limitations.

LIMITED DEVICES

Home devices have important discrepancies. In some cases, they can have limited resources. [11]

Bandwidth

Subsystems might have limited bandwidth with their devices. [33]

Energy

Devices might have limited energy supply. In particular, mobile devices are often sensitive to power consumption.

Computational power and memory

Devices might have limited computational power and memory.

FAILURES

Subsystems are subject to unpredictable failures.

3.3.2 Economical

Economical aspects of the system are not studied in depth. However, in order to be adopted, it should respect economical constraints.

Costs

Installation, maintenance, acquisition cost should be minimized. In order to be interesting for users, the system has to be affordable. In particular, it should not require unrealistic hardware. [36, 13]

Business Model

The system should enable vendors to sell new devices and software for the system. Vendors have an interest in adopting the system. [13]

Closed subsystems

Subsystems might only offer partial and/or restricted access to it. For example, subsystems might use patented technologies or exert content restrictions. [26, 13]

Summary

Figure 3.1 on the next page shows a mindmap of the different goals, features, and constraints.

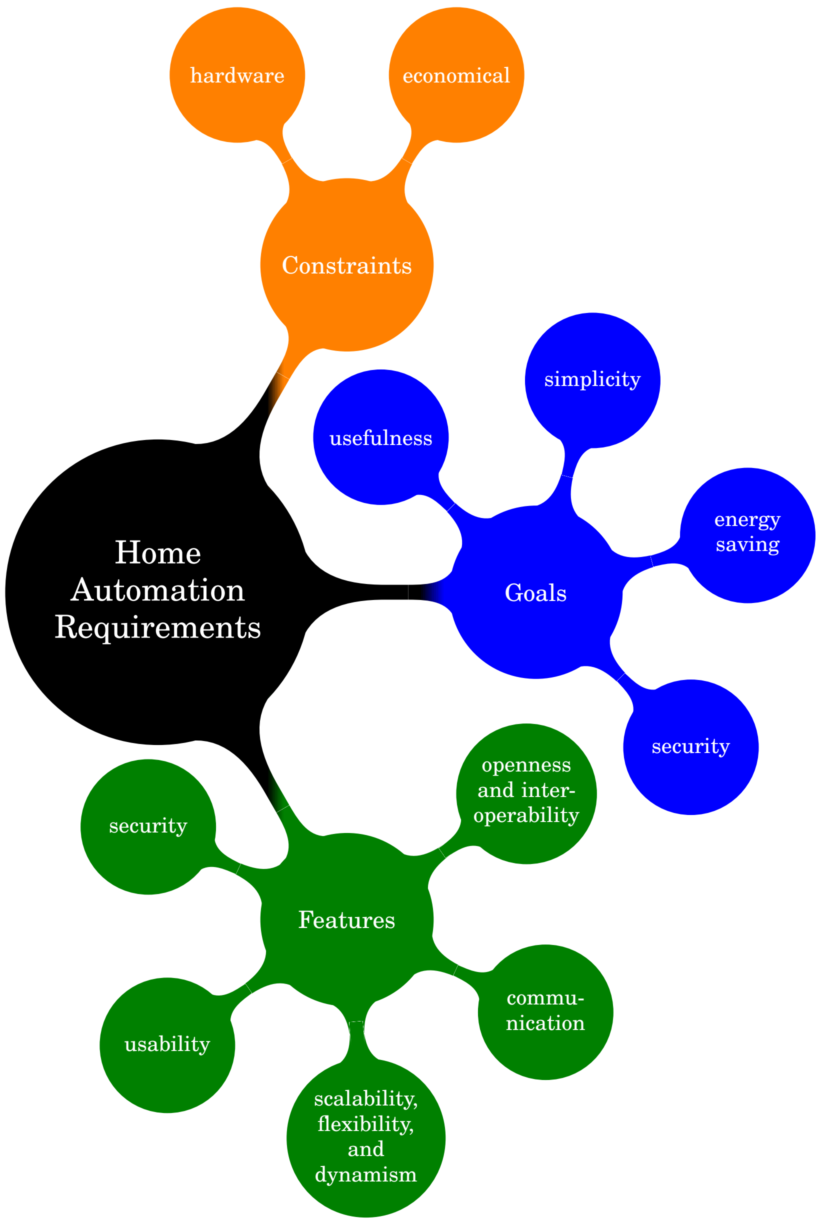


Figure 3.1: Summary of the different requirements of a home automation system.

CHAPTER 4

ANALYSIS AND DESIGN

The following design is based on the design of the HomePort system. (see Section 2.7 on page 29 for more details), various other home automation systems, and open Internet standards. Section 4.1 on the current page presents the different layers of the system and their responsibilities. Afterwards, Section 4.2 on page 48 explains how the layers fulfill the requirements of the system (see Chapter 3 on page 33 for more details). Finally, the layer repartitioning on different device components is shown in Section 4.3 on page 69. This section also explains subsystem's interactions with layers and message exchanges.

4.1 LAYERED ARCHITECTURE

The architecture of the system is divided into four different layers. The architectural layering has sought properties:

- clear responsibilities for each layer
- low layer coupling
- simpler and smaller individual components
- standard layer-communication interface
- layer independence (from vendor, programming language, implementation, ...)
- a layer implementation can be seamlessly replaced by another one.

This is a traditional approach to building complex systems such as the OSI model (see Section 2.2 on page 17 for more details).

4.1.1 OVERVIEW

The system has four layers. From the lowest level of abstraction to the highest, these layers are: the device layer, bridging layer, service layer, and composition layer. Layers

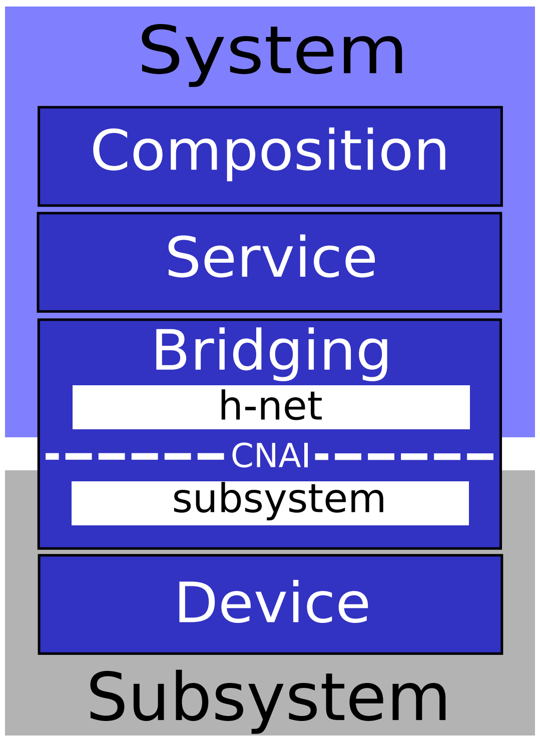


Figure 4.1: Overview of the architectural layers of the system.

are spread from subsystems to the home automation system (the bridging layer being the border between the two).

Figure 4.1 on this page presents an overview of the different architectural layers of the system. Moreover, it shows how the layers are partitioned between the subsystem and system level.

4.1.2 Device Layer

The device layer is responsible for controlling devices from a specific subsystem. Therefore, it is implemented at the subsystem level by its vendor. It contains a set of devices connected between them by the subsystem protocols.

The design of the system makes no assumptions on the devices or the subsystem protocols. However, the subsystem has to offer the possibility to communicate with it. It is the vendor's choice to decide how to pass messages from and to the bridging layer.

4.1.3 Bridging Layer

The bridging layer is responsible for making available subsystem devices over an IP network. In order to do so, it translates messages from the subsystem specific protocols to TCP/IP or UDP/IP (see Subsection 2.3.1.1 on page 21 for more details) from the service layer, and vice-versa.

It has the advantage that it does not require substantial changes in the system. It merely needs one more device component to interact with the other subsystem devices. Besides, the translation is done at a low level and TCP/IP or UDP/IP have low resource demands. However, it requires that the vendor produces an additional device with an Ethernet network card to implement the layer.

The layer is it-self split in two. One sublayer that is subsystem dependent and another that is generic for the LAN communication.

4.1.3.1 Subsystem

This sub-layer is responsible for translating messages from and to the subsystem devices. Therefore, it has to intercept messages that are intended for devices outside the subsystem control. Additionally, it injects messages coming from the service layer in the device layer, as if they were coming from a subsystem device.

4.1.3.2 Heterogeneous Network Protocol

This sub-layer is responsible for communicating messages over Ethernet. It is only made of generic and standard components. If necessary, it ensures bridging/service layer discovery and configuration, and IP level security (see Section 2.6.3 on page 28 for more details).

4.1.3.3 Common Network Adaptor Interface (CNAI)

The common network adaptor interface (CNAI) is the interface that separates the two bridging sublayers. It defines how they communicate. Moreover, it enables total translation freedom at the subsystem level and the use of a generic heterogeneous network protocol.

4.1.4 SERVICE LAYER

The service layer is responsible for presenting devices in a uniform and subsystem independent manner. It exposes devices from the device layer as resources. Resources can be manipulated using HTTP (see Subsection 2.3.1.2 on page 21 for more details) through a REST architecture (see Section 2.5.2 on page 24 for more details). The layer enforce security policies between subsystems on devices. In addition, it enables discovery of controlled resources (see Section 2.5.4 on page 26 for more details) for the composition layer.

4.1.5 COMPOSITION LAYER

The composition layer is responsible for combining devices presented by the service layer. Interactions between devices of different subsystems are programmed at this layer. They can be hand-coded or programmed with a service composition language (such as HCLL [13] or a finite-state machine based language).

Layer	Responsibilities
Composition	interactions between devices (at the system level)
Service	presentation and enforcement of security policies on the devices (at the system level).
Bridging	message passing and translation between the system and the subsystems.
Device	control of the devices (at subsystem level).

Table 4.1: Summary of layer responsibilities of the system.

Summary Of Layer Responsibilities

Table 4.1 on this page summarizes the responsibilities of the different architectural layers of the system.

4.2 Facets Of the System

This section explains how the system works in its three main aspects:

- the automatic discovery and configuration in Section 4.2.1 on the current page
- the communication in Section 4.2.2 on page 53
- the security in Section 4.2.3 on page 62
- the redundancy and scalability in Section 4.2.4 on page 64

4.2.1 Automatic Discovery and Configuration

The goal of the automatic discovery and configuration is to provide the information needed for layers to work. The system layers can be implemented on different devices. Furthermore, devices at the system level are connected between them by an Ethernet network. Therefore, dependent layers need ways to contact each other.

It is assumed that devices of the system have their IP addresses already attributed. There are well known ways to automatically assign IP addresses to devices of an Ethernet network such as DHCP or ZeroConf.

4.2.1.1 Simple discovery protocol

The following is a simple protocol for devices implementing the bridging layer (bridges) to discover the IP addresses of devices implementing the service layer (gateways).

1. A bridge multicasts an UDP/IP packet on a defined port to request devices implementing the service layer. In order to allow new versions and functionalities, it should request a minimal version number of the service layer in the message.
2. A gateway, implementing at least the version number requested, replies with its IP address. The gateway can decide to reply only to the bridge that made the request or by broadcast, in order to tell the other bridges that could be interested. In the reply, the gateway should specify the version of the service layer that it implements.

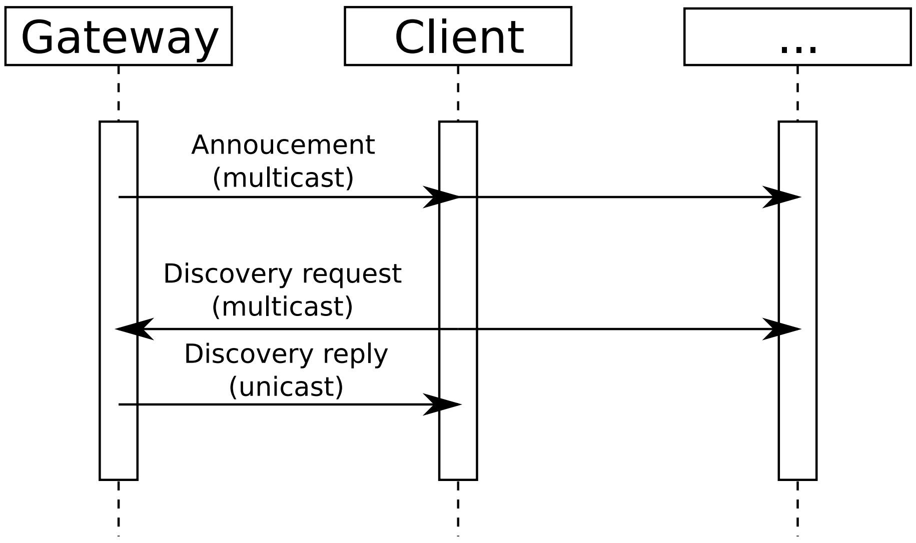


Figure 4.2: Gateway discovery process with a simple protocol.

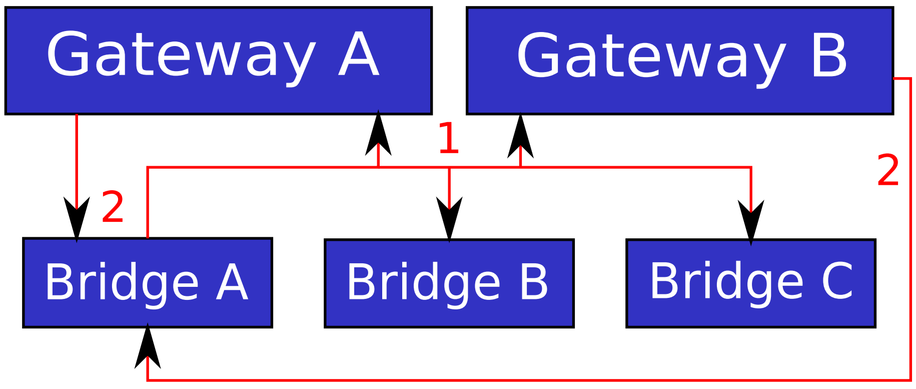


Figure 4.3: Example of a bridge discovering gateways of the system.

3. Finally, the bridge can select one (or more) gateways to communicate with. It can decide based on the response time, the version number, or any adequate criteria.

Figure 4.2 on the current page shows the process by which a gateway is discovered by clients. Firstly, the gateway announces periodically its presence with a message. It is similar to a discovery reply message, except than it is sent in multicast. Secondly, a client sends a discovery request and the gateway replies.

Example:

Figure 4.3 on the preceding page shows an example of the protocol for bridges to discover gateways in the system.

1. `Bridge A` broadcasts a request to all components connected to the network.

2. The two gateways reply to `Bridge A`.

`Bridge A` can then interact with both gateways or select the one that fit its needs the best.

This simple protocol leaves a maximum of freedom to bridges and gateways, while still achieving its goal.

There exists alternative ways to carry out automatic gateway discovery, such as using a DNS, the UPnP (Simple Service Discovery Protocol (SSDP)) protocol (see Section 2.3.2 on page 22 for more details), the Service Location Protocol (SLP) [23], ...

4.2.1.2 Devices Profile for Web Services

There are existing standards that meet many of the system requirements. One of them is "Devices Profile for Web Services" (DPWS), a successor of UPnP. Its goals are: [29]

- sending secure messages to and from a web service
- automatically discovering a web service
- describing a web service
- subscribing to, and receiving events from, a web service

DPWS is more than merely an evolution of the UPnP standard. It uses the service-oriented paradigm. To achieve its goals, it combines protocols introduced previously. It uses HTTPS to send secure messages (see Section 2.6.4 on page 28 for more details). To automatically discover web services, it takes advantage of techniques presented in Section 2.5.4 on page 26, and in particular of WS-Discovery. Web services are described with WSDL (see Section 2.5 on page 24 for more details). Finally, the eventing functionality enables subsystems to subscribe to relevant events. This functionality is assured by WS-Eventing [48].

WS-DISCOVERY

The WS-Discovery protocol supports two modes: ad hoc and managed. The system does not make any supposition about explicit network management services (such as DHCP, DNS, domain controllers, directories, ...). Therefore, the ad hoc mode is the most appropriate. In this mode, the client (a controller or a bridge) sends a multicast (see Subsection 2.3.1.1 on page 21 for more details) probe to find a target service (the gateway service layer). The gateway that matches the probe replies to the client. Finally, the client can locate the service by sending a multicast message resolve request. When the gateway leaves the network, it tries to send a multicast bye message to inform clients. The protocol generates a lot of multicast messages. In order to minimize the amount of such messages,

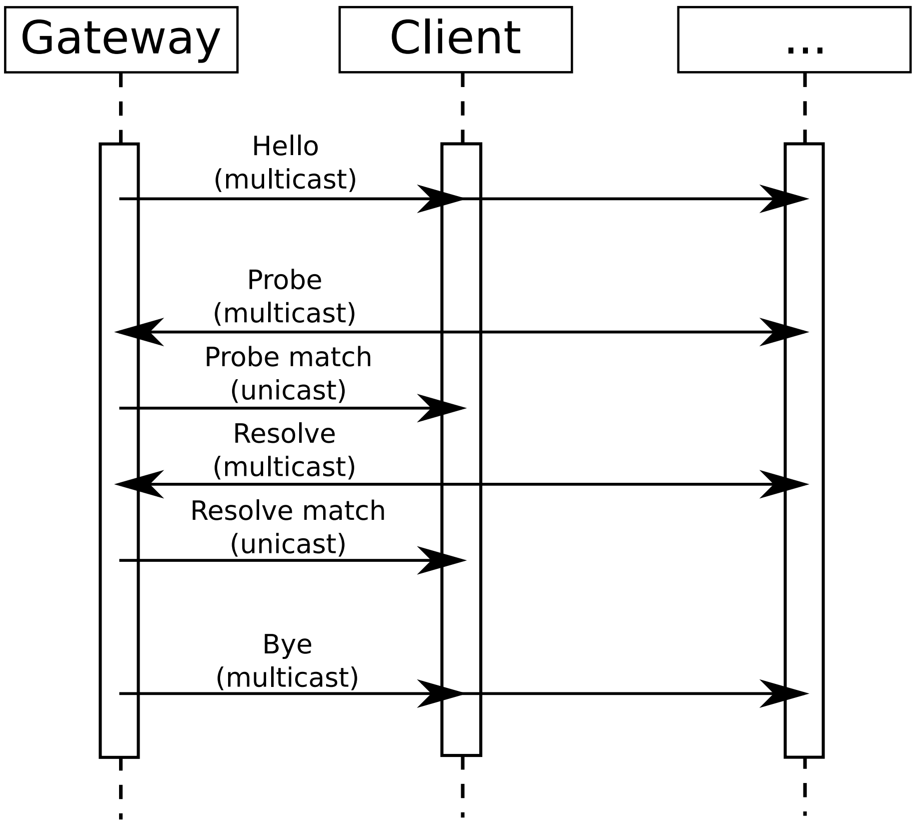


Figure 4.4: WS-Discovery protocol applied to a gateway.

the gateway can send a multicast service announcement message that can be detected by clients. This minimizes the total amount of multicast messages exchanged, by sending announcements to possibly more than one client at the time.

Figure 4.4 on this page shows the WS-Discovery process for a gateway. It is a new service added that did not exist in the initial HomePort system.

Probe

Listing 4.1 on the current page presents a probe message to discover services offered by gateways.

Listing 4.1: WS-Discovery: Probe message

```
<s:Envelope ... >
  <s:Header ... >
    <a:Action ... >
      http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Probe
    </a:Action>
    <a:MessageID ... >xs:anyURI</a:MessageID>
   [<a:ReplyTo ... >endpoint-reference</a:ReplyTo>]?
    <a:To ... >xs:anyURI</a:To>
    ...
  </s:Header>
  <s:Body ... >
    <d:Probe ... >
     [<d:Types>list of xs:QName</d:Types>]?
     [<d:Scopes [MatchBy="xs:anyURI"]? ... >
        list of xs:anyURI
      </d:Scopes>]?
     ...
    </d:Probe>
  </s:Body>
</s:Envelope>
```

- `/s:Envelope/s:Header/*` contains metadata to process the message.
- `/s:Envelope/s:Body/d:Probe/d:Types` defines the type of target service requested. The application defines a scheme. The following types could be used:
  - *wse:eventing*: the event notification service
  - *gateway:device*: the resource manipulation service
  - *gateway:selector*: the group identifiers and functionality dependencies service

  The system can be extended by defining new types.

4.2.1.3 Group identifiers and functionality dependencies

Devices can have group identifiers and functionality dependencies associated with them. These identifiers and dependencies enables to select a set of devices. These selectors can be defined for a device or an existing selector. See Section 3.2.3 on page 37 for more details.

It is possible to select the intersection, union, and the difference of a set of devices with the `AND`, `OR`, `NOT` operators. The three operators form a complete set of logical connectives (expressively adequate). Both kind of selector supports the single-inheritance principle. Therefore, the inheritance can be represented as a tree data-structure. Once an element of the tree is selected, all its children are also selected.

Finally, once devices are registered, a gateway can provide a list of devices based on group identifiers and functionality dependencies associated with them.

Listing 4.2 on the next page describes the WSDL interface of this service. It is a new service added that did not exist in the initial HomePort system.

Example:

A device is registered with request such as:
`https://gateway1/register?identifier=0x23f0d9`
`&group=kitchen,light&functionality=dimmer`
A request such as:
`https://gateway1/selector?group=kitchen AND light`
`&functionality=dimmer`
would provide a list of registered devices belonging to the group identifiers "kitchen" and "light", and that have functionality dependency "dimmer".

Listing 4.2: WSDL of the gateway group identifiers and functionality dependencies

```
<wsdl:binding name="DeviceServiceHttpBinding"
 whttp:methodDefault="GET"
 interface="gateway:ServiceInterface"
 type="http://www.w3.org/ns/wsdl/http">
    <wsdl:operation ref="gateway:getDevicesIdentifiers"
      whttp:location="/list?identifier={identifier}">
        <wsdl:outfault
         ref="gateway:IdentifierNotFoundException"/>
    </wsdl:operation>
    <wsdl:operation ref="gateway:registerDevicesIdentifiers"
      whttp:location="/register?identifier={identifier}&group={group}&
           functionality={functionality}">
        <wsdl:outfault
         ref="gateway:IdentifierNotFoundException"/>
        <wsdl:outfault
         ref="gateway:GroupNotFoundException"/>
        <wsdl:outfault
         ref="gateway:FunctionalityNotFoundException"/>
    </wsdl:operation>
    <wsdl:operation ref="gateway:selectDevices"
      whttp:location="/selector?group={group}&functionality={
           functionality}">
        <wsdl:outfault
         ref="gateway:GroupNotFoundException"/>
        <wsdl:outfault
         ref="gateway:FunctionalityNotFoundException"/>
        <wsdl:outfault
         ref="gateway:LogicalOperatorsException"/>
    </wsdl:operation>
    <wsdl:fault ref="gateway:IdentifierNotFoundException"/>
    <wsdl:fault ref="gateway:GroupNotFoundException"/>
    <wsdl:fault ref="gateway:FunctionalityNotFoundException"/>
    <wsdl:fault ref="gateway:LogicalOperatorsException"/>
</wsdl:binding>
```

4.2.2 COMMUNICATION

This part describes the different means available to the components of the system to communicate. Subsection 4.2.2.1 on this page describes methods to manipulate devices. In addition, Subsection 4.2.2.2 on page 56 presents the notification protocol for events. Finally, Subsection 4.2.2.3 on page 60 defines a data format to interact with devices.

4.2.2.1 REST architecture

A REST architecture style is enough for subsystems to communicate, as described in the HomePort system (see Section 2.7 on page 29 for more details). It is an efficient way to

Method	Action
`GET /devices`	get a list of resources *devices*.
`GET /devices/identifier`	show a representation of the resource *identifier*.
`POST /devices`	create a representation of the resource based on the data posted.
`PUT /devices/identifier`	update a representation of the resource *identifier* based on the data sent.
`LISTEN /devices/identifier`	subscribe to the simple event notification system (from HomePort, not HTTP).
`DELETE /devices/identifier`	delete the resource *identifier* representing a device.

Figure 4.5: REST methods associated with the actions in the system.

increase the system generality and scalability, as described in Section 2.5.2 on page 24.

Subsystem devices are represented as resources. A subsystem can access information about another subsystems' devices by accessing the resources that represent them. Moreover, a subsystem can manage devices it controls by modifying resources that represent them. All available devices can be listed with a link to the resource. Figure 4.5 on the current page describes REST-style HTTP methods of the system.

The service h-net sub-layer supports five request methods:

- `GET`: access the representation of a resource of the system (e.g. a device)
- `POST`: create a resource with a representation of a device
- `PUT`: update a resource with a representation of a device
- `LISTEN`: subscribe to the notification of the change of a resource.
- `DELETE`: delete a resource.

This interface respects the HomePort system interface. The subsystem sub-layer has to use these general methods to translate subsystem commands.

Example:

The changes of a switch is monitored by sending a *listen request* to the bridge or the switch. When a change happens, a token is sent to the client and a *get request* is sent by the client to know the new state of the resource. Based on the new state, a light can be turned on by sending a *put request* to the bridge controlling the light.

`GET` and `PUT` requests enable to read and modify any kind of resource represented on the gateway. They have the advantages of a REST architecture. In particular, they are easy to implement and have low encapsulation overhead.

WSDL GATEWAY DESCRIPTION

The WSDL gateway description of the system (Listing 4.3 on this page) defines how clients can request the gateway resources. It is the first strict definition of the HomePort interface.

Listing 4.3: WSDL of the gateway

```
<wsdl:description xmlns:wsdl="http://www.w3.org/ns/wsdl"
   targetNamespace="http://www.homeport.org/gateway/device/wsdl"
   xmlns:gateway="http://www.homeport.org/gateway/device/wsdl"
   xmlns:whttp="http://www.w3.org/ns/wsdl/http"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:msg="http://www.homeport.org/gateway/device/xsd">

<wsdl:binding name="DeviceServiceHttpBinding"
 whttp:methodDefault="GET"
 interface="gateway:ServiceInterface"
 type="http://www.w3.org/ns/wsdl/http">

    <wsdl:operation ref="gateway:getDevices"
     whttp:location="devices/{identifier}">
        <wsdl:outfault
         ref="gateway:DeviceNotFoundException"/>
    </wsdl:operation>
    <wsdl:fault ref="gateway:ExistingDeviceException"/>
    <wsdl:fault ref="gateway:InvalidDataException"/>
    <wsdl:operation ref="gateway:updateDevice"
     whttp:location="devices"
     whttp:method="POST">
        <wsdl:outfault
         ref="gateway:DeviceNotFoundException"/>
    </wsdl:operation>
    <wsdl:fault ref="gateway:DeviceNotFoundException"/>
    <wsdl:fault ref="gateway:InvalidDataException"/>
    <wsdl:operation ref="gateway:updateDevice"
     whttp:location="devices/{identifier}"
     whttp:method="PUT">
        <wsdl:outfault
         ref="gateway:DeviceNotFoundException"/>
    </wsdl:operation>
    <wsdl:operation ref="gateway:listenDeviceEvent"
     whttp:location="devices/{identifier}"
     whttp:method="LISTEN">
        <wsdl:outfault
         ref="gateway:DeviceNotFoundException"/>
    </wsdl:operation>
    <wsdl:operation ref="gateway:deleteDevice"
     whttp:location="devices/{identifier}"
     whttp:method="DELETE">
```

```
        <wsdl:outfault
          ref="gateway:DeviceNotFoundException"/>
        <wsdl:outfault
          ref="gateway:ExistingDeviceException"/>
        <wsdl:outfault
          ref="gateway:InvalidDataException"/>
    </wsdl:operation>
    <wsdl:operation ref="gateway:getDevices"
      whttp:location="devices/"/>

</wsdl:binding>
</wsdl:description>
```

HTTP MECHANISMS

The HTTP contains many mechanisms that can be used for the system. It is developed to enable caching, authentication, content compression, partial retrieval of content, content integrity verification, ... In particular, in case of concurrent modification of a resource with *PUT methods*, the content mismatch can be detected by the client with headers such as *If-Match* or *If-Unmodified-Since*. Data transfer are also more efficient as the content is downloaded only if it was modified since the last download.

Content negotiation

The HTTP allows parties to negotiate the content of a resource. Therefore, a resource can have various representations. It enables clients to ask for the representation that fits its need the best. For example, the gateway could provide an XML, plain text, and HTML representation of a resource. The server does not have to keep the different representations. It can store only one and dynamically transform it into another at the client request[1].

4.2.2.2 Event notification

An event notification protocol is used to notify the components of the system of an event happening on a resource (a device). Consequently of an event notification, the component decides to take an action based on the change of the resource.

SIMPLE HOMEPORT NOTIFICATION

The `LISTEN` request is more advanced than the `GET` or `PUT` requests. Indeed, the event notification is a complex task in a distributed system. The current HomePort system does not take into account the following cases:

- unsubscribe to the event notification
- the failure of the bridge sending the notifications
- subscribe to events on behalf of another bridge

[1]XSLT is a language that can transform XML documents into any text-based representations, such as another XML format, HTML, plain text, etc.

- have selective event notifications[2]

The advantage of the current system is that it is simple to implement as it ignores some cases. It also offers the possibility to broadcast a UDP notification (see Subsection 2.3.1.1 on page 21 for more details) of an event. However, in order to have a rich and efficient system, such cases might need to be handled.

WS-EVENTING

WS-Eventing is a standard protocol for web services to subscribe to events and to send notifications. It offers the possibility to solve cases neglected by the current *LISTEN notification system*. It is a new service added that did not exist in the initial HomePort system. It uses SOAP (see Section 2.5.3 on page 25 for more details) to fulfill its objectives. WS-Eventing supports the following subscription messages:

- Subscribe
- Renew
- GetStatus
- Unsubscribe
- SubscriptionEnd

The event sources are naturally the bridges. The gateway is the subscription manager. Finally, every components of the system can subscribe to events (in particular bridges). The protocol does not specify the notification. It can be any kind of message.

Figure 4.6 on the next page shows the WS-Eventing process to notify of changes in devices controlled by a bridge (the source).

Subscribe

To create a subscription, a subscriber sends a request message to an event source (a bridge). Listing 4.4 on page 59 defines the form of such a message. [48]

- `/s:Envelope/s:Header/wsa:Action` is an URI to bind SOAP.
- `/s:Envelope/s:Body/*/wse:EndTo` describes where to send the *SubscriptionEnd* message.
- `/s:Envelope/s:Body/*/wse:Delivery` gives the delivery destination and mode.
- `/s:Envelope/s:Body/*/wse:Expires` defines the expiration time of the subscription.
- `/s:Envelope/s:Body/*/wse:Filter` filters an event source in a defined dialect[3].

[2]In order, to have targeted and efficient notifications.
[3]A dialect is a language to filter events (e.g. XPath describes a path in XML document).

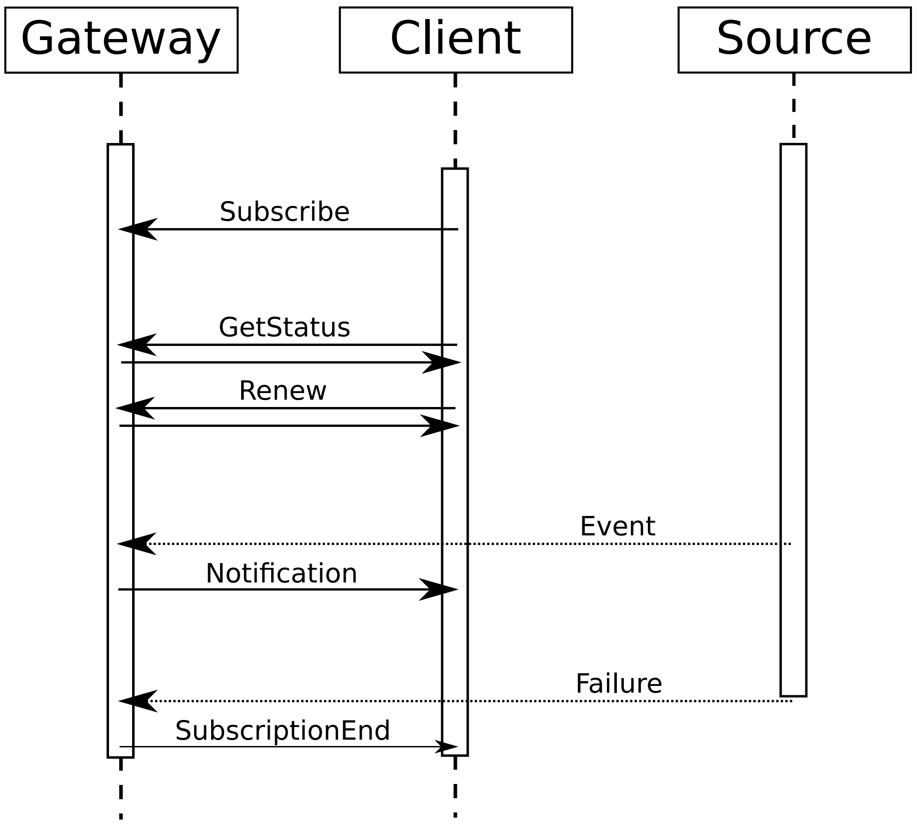


Figure 4.6: WS-Eventing protocol applied to notify of changes in devices.

Listing 4.4: WS-Eventing: Subscribe message

```
<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://gateway/event/Subscribe
    </wsa:Action>
    ...
  </s:Header>
  <s:Body ...>
    <wse:Subscribe ...>
      <wse:EndTo>endpoint-reference</wse:EndTo> ?
      <wse:Delivery Mode="xs:anyURI"? >xs:any</wse:Delivery>
      <wse:Expires>[xs:dateTime | xs:duration]</wse:Expires> ?
      <wse:Filter Dialect="xs:anyURI"? > xs:any </wse:Filter> ?
      ...
    </wse:Subscribe>
  </s:Body>
</s:Envelope>
```

Renew

The `Renew` message is used to extend an expiring subscription. Listing 4.5 on the current page defines the form of such a message. [48]

Listing 4.5: WS-Eventing: Renew message

```
<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://gateway/event/Renew
    </wsa:Action>
    ...
  </s:Header>
  <s:Body ...>
    <wse:Renew ...>
      <wse:Expires>[xs:dateTime | xs:duration]</wse:Expires> ?
      ...
    </wse:Renew>
  </s:Body>
</s:Envelope>
```

- `/s:Envelope/s:Header/wsa:Action` is an URI to bind SOAP.
- `/s:Envelope/s:Body/*/wse:Expires` defines the new expiration time of the subscription.

GetStatus

The `GetStatus` is used to receive the status of the subscription. The subscriber sends a request to the subscription manager (the gateway).

Unsubscribe

Even if subscriptions end, a subscriber might want to unsubscribe earlier to an event. The `Unsubscribe` message provides this possibility. After the message is received, the subscriber will not receive notifications for the specified event.

SubscriptionEnd

If the subscription source (a bridge) fails, the subscriber is notifier by the subscription manager with a `SubscriptionEnd` message. Therefore, it has the possibility to react accordingly.

4.2.2.3 Data exchange format

This part proposes basic XML representations of resources that contains accessible data about devices. Other representations could be offered to clients by the virtues of the HTTP content-negotiation and XML transformation-language mechanisms. It is a conventional format that did not exist in the initial HomePort system.

STATE

Listing 4.6 on this page presents a basic XML representation of the state of a device. The content of the device that is accessible to the system is represented in this resource. It can then be manipulated with the request methods of the REST architecture of the system.

Listing 4.6: Basic XML representation of the state of a device.

```
<states type="array">
  [<state>
    <name>...</name>
    [<read-only-state type="boolean">true|false</read-only-state>]
    <state-type>..</state-type>
    [<description>[...]</description>]
    <value encoding="xs:any">...</value>
    [<created-at type="xs:any">[...]</created-at>]
    [<updated-at type="xs:any">[...]</updated-at>]
  </state>]
  [...]
</states>
```

- `/states/*` represents all states of the device.
- `/states/state*` represents the state of the device.
- `/states/state/name/` defines a unique name for the content.
- `/states/state/readonly/` defines if the content can be modified (false by default).
- `/states/state/state-type/` defines how to interpret the content.
- `/states/state/description/` describes the content.

- `/states/state/value/` is the current value of the content.
- `/states/state/value/@encoding` defines how the value is encoded.
- `/states/state/created-at/` is the time representation when the state was created.
- `/states/state/updated-at/` is the time representation when the state was last updated.

The content is customizable for vendors by defining new types. In particular, the value of the content can store any kind of data (text, XML, base-64 encoding, ...).

METADATA

Listing 4.7 on the current page defines metadata that do not change as often as the state of the device. It is separated to make state transfer faster.

Listing 4.7: XML Metadata of a device.

```
<device>
  <identifier>...</identifier>
  [<comment>[...]</comment>]
  [<status>[...]</status>]
  [<created-at type="xs:any">[...]</created-at>]
  [<updated-at type="xs:any">[...]</updated-at>]
  <states type="array">
    [<state>
      <name>...</name>
      <state-type>...</state-type>
    </state>]
    [...]
  </states>
</device>
```

- `/device/identifier/` is a unique device identifier.
- `/device/comment/` describes the device.
- `/device/status/` describes the current status of the the device.
- `/device/created-at/` is the time representation when the device was created.
- `/device/updated-at/` is the time representation when the device was last updated.
- `/device/states/*` is the list of states of the device with its `name` and `state-type` (see data exchange format for state).

4.2.3 Security

This part describes different security aspects of the system. First, it defines the security perimeter of the system. The following two sub-parts present, layer by layer, the security techniques to ensure confidentiality and integrity, and authentication and authorization. Finally, the last part analyzes and suggests solutions to ensure the availability of the system.

4.2.3.1 Security perimeter

The system enforces security where it controls the information exchanged.

Therefore, it does not offer any kind of security at the subsystem level. Most subsystems have mechanisms to control the inclusion of new devices in the subsystem, such as Bluetooth with the pairing or Wi-Fi with WPA (see Section 2.2.2 on page 18 for more details).

Depending on the technology used and the kind of information exchanged, the subsystem encrypts data exchanged with its devices or not. If the subsystem does not provide any confidentiality and integrity mechanisms, it does not make sense to use cryptographic techniques to communicate with this subsystem. Indeed, messages encrypted at the system level could be easily intercepted or manipulated at the subsystem level. Therefore, a subsystem can decide not to use encryption in the system. However, messages coming from a secure subsystem can not be transmitted to another insecure subsystem.

The service layer enforces the access control between subsystems on the devices. Therefore, the lower bridging layer only provides data confidentiality and integrity.

4.2.3.2 Confidentiality and integrity

Subsystem and bridge

Many solutions exists to ensure the confidentiality and integrity of messages exchanged between the subsystem and the bridge.

IPsec offers a low level and well supported solution (see Section 2.6.3 on page 28 for more details). It enables to secure the communications from end-to-end between subsystems and bridges. IPsec is merely an additional layer. Therefore, it does not add any constraints on the communication protocol between the subsystem and the bridge. The addition of the security layer is transparent for upper layers. Finally, if the bridge is directly integrated in the subsystem, this security layer might not be needed.

Bridge, gateway, and controller

One of the goals of the gateway is to enforce security policies. In order to achieve this requirement, it is necessary for the gateway to trust the content of transactions. The confidentiality and integrity ensure that transactions remain secret and unmodified. This is possible by using IPsec in the same way as between the subsystem and the bridge. However, in order to treat trust as a whole, a security layer acting at an upper level is needed. HTTPS offers such a layer. It integrates well with web services, as described by WS-Security (see Section 2.6.5 on page 29 for more details). Although, it is independent from this standard.

By using certificates for bridges, gateways, and controllers, secure channels can be established between parties.

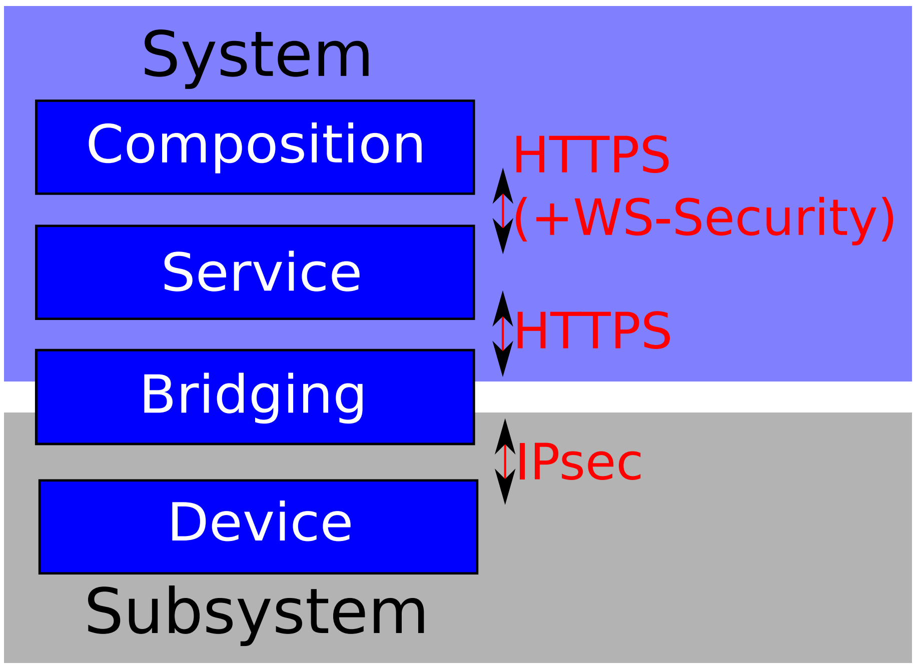


Figure 4.7: Overview of the encryption protocols used between layers of the system.

4.2.3.3 Authentication and authorization

As explained previously, the trust between the various components of the system is only assured from the service layer to the composition layer.

Subsystem and bridge

The bridge authenticates communications coming from the subsystem that it is programmed to collaborate with. The Internet Key Exchange (IKE or IKEv2) protocol, which is part of IPsec, permits this authentication. It can be done with a shared key between the subsystem and the bridge or with public keys. This does not require any interaction from the user. As for the confidentiality and integrity part, it might not be required if the bridge is directly integrated in the subsystem.

Bridge, gateway, and controller

For the higher layers, the identity of the components of the system is verified by certificates. Each component (bridges, gateways, and controllers) has a certificate signed by a trusted third party. In order to keep the system open to all vendors, users are free to modify the list of trusted third parties.

At this point, the user has to interact with the system to authorize new components in the system. Even if the component is validated by a trusted third party, the genuine user might not want a component to be added to his system. Indeed, if it was not the case, an intruder could buy a validated component that the intruder controls and add it seamlessly to the system without the genuine user's knowledge. Therefore, it could control the devices of the system. To avoid this situation, the user has to authorize, through the gateway, components to be added. The gateway could present to the user the serial number or hashed value of the component certificate. The user could then possibly compare it to a number printed on the component. This interaction can be done through a web interface on the gateway, a customized graphical user interface, or a physical interface (such as a touch screen).

The gateway keeps a list of system components validated by the user. Thereafter, these components with valid certificate are trusted. The user is free to modify the list of trusted components.

4.2.3.4 Availability

The gateway is a critical part of the system. Indeed, it stores essential data about the whole system. In contrast, the bridge controls only a particular subsystem and the controller composes only part of the system logic. More than one gateway can be present in the system. However, something has to be done for a gateway to take over in case of failure.

A simple scenario would be for bridges and controllers to discover new gateways when the original one is not responding after a certain time. Consequently, bridges can register their devices and controllers can combine them, based on the information stored on the new gateway. Such a scenario might seem cumbersome and resource consuming, but the failure of a gateway is a rare event. Additionally, bridges and controllers can collaborate with all available gateways. Therefore, in case of the failure of a gateway, no additional steps has to be taken.

In order to avoid denial of service attacks, the gateway should ignore messages coming from untrusted components. In consequence, the amount of resource needed to treat untrusted requests is greatly reduced. Ergo, it limits the impact of such an attack.

Section 4.2.4 on the current page presents a more advanced scenario to ensure the availability of the system through mechanisms of redundancy and scalability.

4.2.4 Redundancy and Scalability

In order to ensure the redundancy and scalability of the system, many aspects have to be taken into consideration. Obviously, the direct communication between devices of different subsystems has to scale to the size of the system (see Section 2.4.2 on page 23 for more details). Additionally, redundancy mechanisms have to be provided in the systemDS-Redundancy. However, other aspects have to be considered. The discovery and notification services have also to offer the same abilities. Indeed, without those services the ability to communicate between devices is not useful. In particular, the services offered by the gateway have to be distributed on different physical devices in order to offer the possibility to scale to the size of the system and offer higher availability. These possibilities are described in Subsection 4.2.4.1 on the current page. Additionally, failures at the bridge level have to be handled properly. This is described in Subsection 4.2.4.2 on the facing page.

4.2.4.1 Gateway

In order for the service layer to be able to scale and having redundancy, the services provided by the layer have to be distributed over more than one gateway. Therefore, it is necessary to define a protocol for the gateways to communicate and keep a coherent representation of the system, in particular in case of failures.

When a bridge connects to the system, it selects a gateway to communicate with, at the end of the discovery protocol. The selected gateway will be responsible for the future communications with that subsystem. A bridge is free to use any non-static method to select its gateway[4]. Additionally, it advertises to the other gateway it is responsible for the devices of the subsystem[5].

The responsible gateway communicates with the bridge to know the current state of the devices controlled by the bridge. At the end of this step, the gateway can present the devices with the REST interface (see Subsection 4.2.2.1 on page 53 for more details) to the rest of the system.

At this point, one could suggest to keep a distributed high-consistency state of the devices on the gateways of the system. In case of failure, the state could be recovered and a new responsible gateway be elected. However, this approach is communication-wise expensive. In particular, in the case of small or frequent changes in the state of a device, the communication overhead between gateways is high. Therefore, this solution scales poorly.

The failure of a gateway is detected when a subsystem for which it is responsible or the other gateways fail to communicate with the gateway. This is known as lazy or on-demand state reaggregation. When a subsystem detects the failure of its gateway, it selects a new responsible gateway with the discovery mechanisms. In the second case, when another gateway detects the failure of the responsible gateway, it has to wait until the bridge also detects the failure. Indeed, at this point, the bridge is the only one to know the state of the devices that it controls and there is no way to contact it as the IP address of the bridge is stored in the failed gateway. This downtime can be minimized by "pinging" the gateway when no communication happens for a certain interval of time between the bridge and its gateway. Ergo, the bridge can discover the failure of its gateway.

When a gateway receives a request for a device for which it is not responsible it passes the request the responsible gateway. This gateway receiving the request can be seen as an intermediate device in a REST scenario (see Section 2.5.2 on page 24 for more details). This is totally transparent for the bridge that initiated the request by taking advantage of the REST architecture of the system.

This whole approach has the advantage of being simple: it does not require additional protocols and works transparently with the other services of the system.

Figure 4.8 on the following page shows the different steps of the collaboration between gateways pass requests to/from a bridge.

4.2.4.2 Bridge

When a bridge fails the communication with the devices of the subsystem that it controls is physically impossible. Once the gateway responsible for that subsystem tries to send a message to it, it will detect the failure.

Additionally, devices can fail or be unreachable independently. The bridge controlling these devices can detect it and send a `DELETE` request to the gateway for the devices concerned.

[4]A bridge can select the first gateway to reply, randomly select gateway that replied, etc. However, it cannot the load on gateways has to be fairly distributed.

[5]This can be done with WS-Discovery (see Section 2.5.4 on page 26 for more details).

[5]The failure can also be detected by "pinging" the bridge as previously explained.

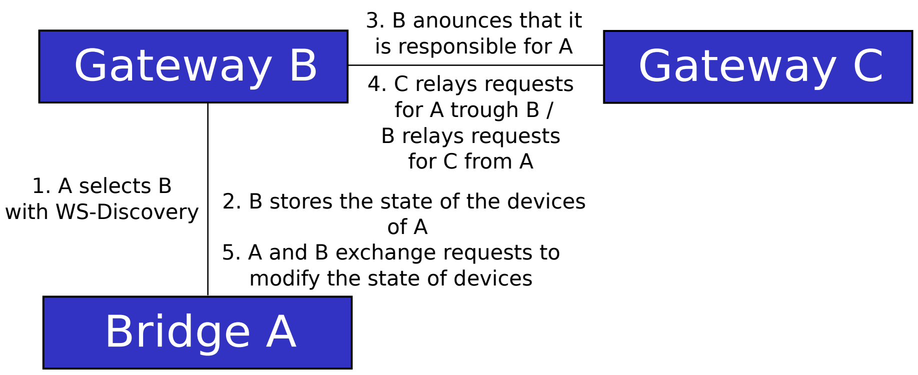

Gateway B
Gateway C
Bridge A
3. B anounces that it is responsible for A
4. C relays requests for A trough B / B relays requests for C from A
1. A selects B with WS-Discovery
2. B stores the state of the devices of A
5. A and B exchange requests to modify the state of devices

Figure 4.8: Overview of the collaboration between gateways B and C to pass requests to/from the bridge A.

When a device or a set of them is unreachable, they are removed from the services offered by the gateway (equivalent to a `DELETE` request). Additionally, the subscription notification is ended for these devices (see Subsection 4.2.2.2 on page 56 for more details). Any request for them to a gateway would return an unavailable status response to the request (see Subsection 2.3.1.2 on page 21 for more details).

Example:

A switch in one subsystem is turned on to activate a light in another subsystem. Lets assume that the two subsystems are registered on two different gateways, in order to generalize this example. The bridge of the switch sends a request to its gateway. The gateway relays the request to the gateway of the light. The light gateway communicates with its bridge to turn on the light. At this point, the bridge detects that the light is unavailable and informs its gateway. The gateway generates an unavailable status response to the initial request that is relayed to the switch. The bridge of the light sends a `DELETE` request to its gateway to address any future request concerning this light. Additionally, any notification request concerning the light is revoked.

Figure 4.9 on the next page shows the minimal setting to ensure the reliability of the communication between the bridge A and B.

Summary

Table 4.2 on the facing page summarizes technologies presented, their purpose, and alternatives. Alternatives are other technologies that were described as possible solutions to the problem.

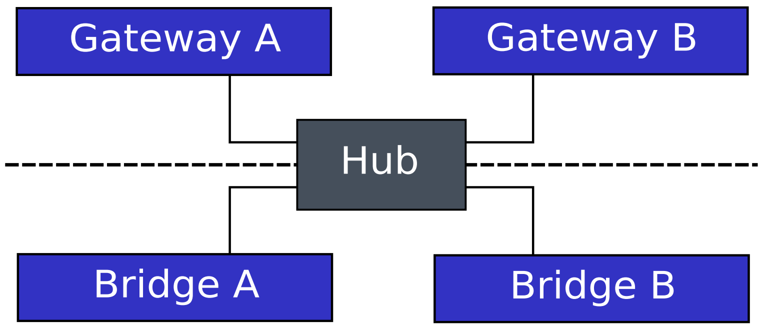


Figure 4.9: Overview of a setting that should ensure the reliability of the system for gateways and bridges.

Purpose	**Technology**	**Alternative(s)**
Attribute IP addresses	DHCP	DNS, ZeroConf, ...
Automatically discover the IP address of components	Simple Discovery Protocol, WS-Discovery	-
Dynamically and automatically discover components	WS-Discovery	-
Select sets of devices based on group identifiers and functionality dependencies	Selector	-
Interact with layers of the system	REST architecture	SOAP, RPC, ...
Provide content-negotiated representations of resources	HTTP mechanisms	*(any other transfer protocol with content negotiation)*
Notify an event	Simple HomePort Notification, WS-Eventing	*(Custom distributed notification protocol)*
Representation of a resource	REST, Data exchange format	*(SOAP, any other XML scheme)*
Integrity and confidentiality of data	IPsec (IKE), HTTPS	WS-Security, ...

Table 4.2: Summary of technologies presented with their purpose and different alternatives.

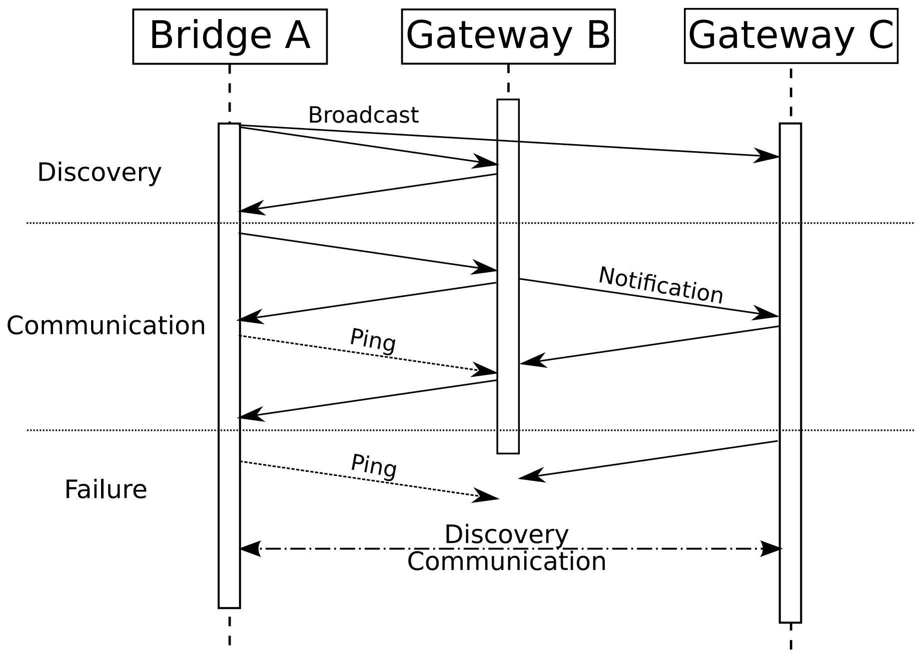


Figure 4.10: Example of message communications in the different aspects of the system.

Example:

Figure 4.10 on the current page shows a scenario of the possible steps between a bridge and and two gateways in the discovery, communication, and failure-detection and recovery phase. In this scenario, the Bridge A starts by sending a broadcast message to select a gateway. Gateway B replies to the request. It is selected as the responsible gateway for A. In the communication phase, A and B exchange messages to modify the state of the devices controlled by the bridge A. Gateway C registered previously to B to be notified of some modification of devices controlled by A. Therefore, gateway B sends a notification to C when such an event happens. It also passes to A messages coming from C. Periodically, A sends a “ping” to B to verify its availability. In the last phase, the scenario shows how A discovers the failure of B by sending a “ping”. Once the failure is detected, A starts a new discovery phase and the communication with C can resume.

4.3 System Interactions

After having discovered available gateways (see Subsection 4.2.1.1 on page 48 for more details), the bridge has to select the gateways with which to register its devices. It provides data about the devices to the selected gateways. Based on the device descriptions, the gateway presents them appropriately to the rest of the system.

Layer	**Imple-menting device**	**Description**
Composition	Controllers	is connected to the LAN and stores the composition programmed.
Service	Gateways	is connected to the LAN.
Bridging	Bridges	has access to the subsystem media and the LAN.
Device	Subsystem devices	controls the subsystem devices and makes interactions possible.

Table 4.3: Summary of the devices implementing the system layers.

DEVICE COMPONENTS

Each layer is conceived to be implemented on specific device components of the system. Although, for economical reasons, some layers can be implemented in the same device. Table 4.3 on the current page shows the distribution of the implementation of the system layers on the devices.

Device components of the system can be combined in the following manner:

- Device and bridge integrated at the device layer.
- Bridge and gateway with the possibility for other bridges to use the gateway.
- Gateway and controller with the possibility for other controllers to use the gateway.

BRIDGING LAYER

During the subsystem binding, a bridge sends a list of devices controlled by its subsystem to a gateway. Afterwards, it translates and relays messages intended for or coming from its subsystem. In the event that it detects that a device is disconnected from the subsystem, it warns the gateways.

Example:

Figure 4.11 on the following page shows an example of bridges interactions with the system. In this scenario, the `device 1` of the `subsystem A` wants to communicate with the `device n` of the `subsystem B`. We suppose that both subsystems have registered their devices. The `bridge A` contacts the common `gateway` on behalf of the `subsystem A`. The gateway replies with the needed information. These two transactions are shown by the *message 1*. Consequently, the two devices are able to communicate as shown by the *message 2*.

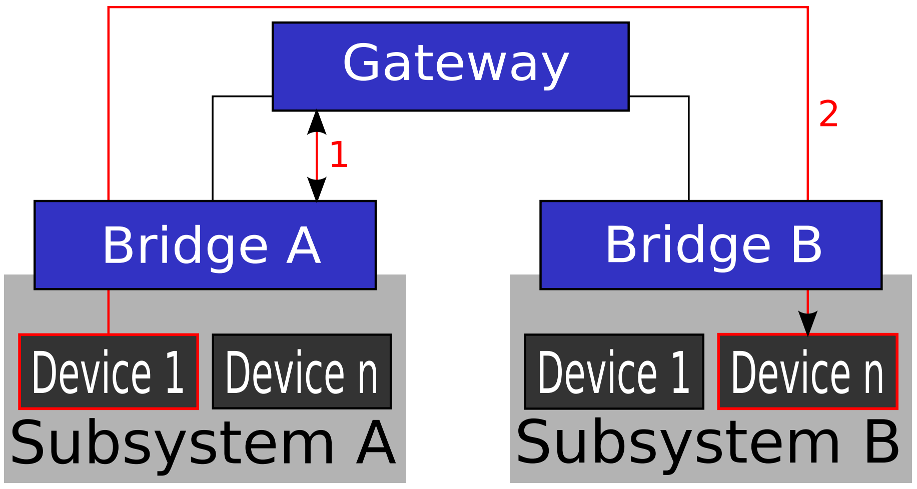


Figure 4.11: Bridges interactions with the system.

Service layer

The gateway receives data about devices from the bridges (trough the h-net protocol). Additionally, it publicized its existence and web service description with WS-discovery. It also provides representations of device resources for the composition layer. If a controller registered with WS-Eventing for an event, the gateway delivers the event to the controller.

Composition layer

As part of the Devices Profile for Web Services (DPWS), WS-Eventing provides for composition web services the possibility to subscribe to particular events happening on the gateway web service. The controllers retrieves representations of device resources from gateways. These representations detail devices for controllers to combine them.

Example:

A controller composing devices that manages lights can be interested in the addition of new devices. Consequently, it subscribes to gateways to receive events concerning the creation of new resources in the "light" category. When such an event happens, it can request the representation of the resource to assess its interest. Based on the decision, it can decide to integrate the new device to the light control.

Summary

Figure 4.12 on the next page shows graphically how the different layers of the system can be combined (from HomePort [13]).

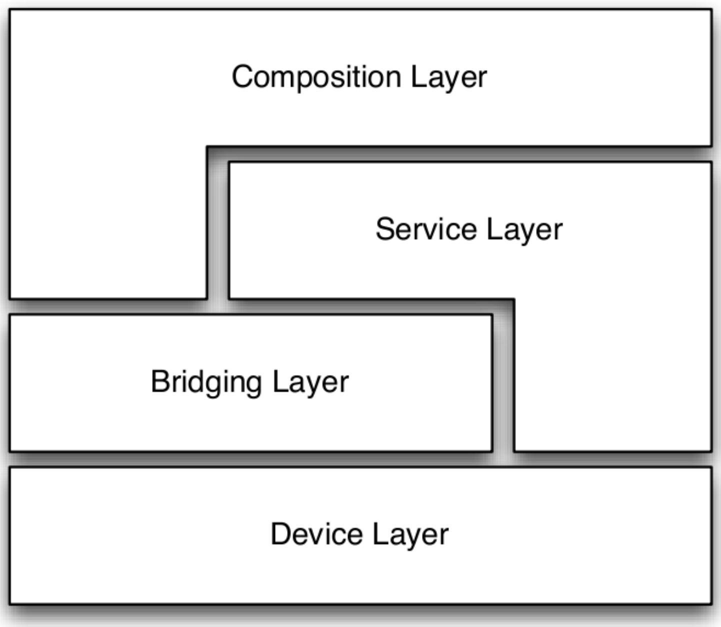


Figure 4.12: Overview of the ways to combine the architectural layers of the system. [13]

CHAPTER 5

IMPLEMENTATION

This chapter describes the implementation of protocols and functionalities defined in Chapter 4 on page 45. It starts by presenting (in Section 5.1 on the current page, and Sections 5.2 and 5.3 on the following page) the context in which the system was developed and the reasoning behind the choices made. Thereafter, it details the implementation in Section 5.5.1 on page 76. It is worth noting that many of the choices made for the implementation are subjective and partially arbitrary. They do not influence deeply the overall functionality of the system.

5.1 OPERATING SYSTEM

Vendors developing a device for a system face four main choices when it comes to run the software or firmware program of the device.

The first and simplest choice is to develop a firmware program that runs directly on the *CPU* of the device, without an operating system (*OS*). If the application is simple and mono-task, it can be an efficient choice. However, it requires the firmware to directly deal with the device resources. Additionally, it does not benefit from any level of abstraction or external libraries. The application might run faster because of the lack of overhead, but it will be harder to port to another device architecture.

A second option is for the vendor to develop an in-house custom *OS*. *OS* offer different levels of abstraction. Additionally, it can also be used for other (future) applications. If devices are very specific, it can be an interesting choice because the *OS* can be tuned for these specific needs. Finally, it offers a high-level of control over the behavior and features of the *OS*.

The third choice is to buy an *OS* made by another company. Many operating systems exist for embedded devices. It eliminates the burden of developing an *OS* in-house. However, it has a cost and it might not support all the needed features.

The last option is to use an open-source *OS*. The most common one is ***Linux***[31], but others exist. It runs on many platforms, but because it is a generic *OS* (not specially developed for embedded devices), it usually requires more resources. However, there exist different *GNU*/*Linux* distributions that are tuned for limited-resource devices. It is free and it can be adapted to the specific needs of the vendor.

The original HomePort implementation was developed for Linux. Therefore, it was a logical choice to develop the new features of the implementation on this OS. It has the advantage of running on many device architectures. In particular, many general purpose device run with this OS, including the NSLU2 (see Section 5.4.1 on the facing page for more details).

Different *Linux* distributions are available for the device. They offer different functionalities. In order to test the implementation, most of them are usable.

5.2 PLATFORM

Linux distributions offer many libraries to facilitate the development of applications. The software can be developed as a script (written in *Python* or *Perl*, for example), a executable binary (written in *C/C++*, for example), or an intermediate language that runs in a *virtual machine* (such as the *Java Virtual Machine* (*JVM*)).

All these options provide rich libraries. Usually, a compiled piece of software is the hardest to port as it needs to be compiled for a specific architecture. Although, libraries provide different levels of architecture-independent abstractions. Scripts are usually slower as they need to be interpreted every time that they are executed. Virtual machines are generally offer a compromise. Languages designed to run in a virtual machine are compiled in an intermediate executable form (called bytecode) and then translated by the virtual machine to the specific architecture of the end-user.

Some language can be run in an *interpreter* as a script or in a virtual machine, such as *Ruby*.

Programs written for the *JVM* are *OS* and architecture independent and they take advantage of caching and just-in-time compilation (although it is also possible for some scripting language). Additionally, it offers the object-oriented paradigm, rich libraries (especially for web services), and *JVMs* specially tuned for resource-limited architecture (such as *Squawk*).

*It was decided to use **Ruby**[15] for most of the development process. It is a general purpose, fully oriented-object language. Additionally, it is intended to maximize developers productivity, and rich web-service libraries exist for it.* As mentioned before, it is possible to compile a Ruby source code to run on the *JVM* (with *JRuby*), or to execute it as a script.

However, the initial *HomePort* program is written in *Python*. It is an older general-purpose programing language by which *Ruby* was inspired.

5.3 LIBRARIES

There exists many web-service (WS) frameworks or libraries for *Ruby* (such as *Camping, Nitro, Sinatra*, ...). ***Ruby on Rails*** (*RoR* or simply *Rails*)[1] is one of these web application framework. It is open source and intended for rapid development. It offers *RESTful* web services. *WS* can easily provide *HTML*, *XML*, or *JSON* output format. See Section 2.3.1 on page 21 for more details.

Rails uses a ***Model-View-Controller*** (*MVC*) architecture pattern. In this architecture pattern, the model stores a domain-specific representation of the data; the view renders the model (for example in *HTML* or *XML*); the controller is the glue-logic dealing with requests and responses and calling models. The goal of the *MVC* is to decrease the coupling between the model and the views, and therefore reduce the complexity and increase maintainability.

Additionally, Rails provides scripts to generate skeleton of models, views, and controllers. Moreover, easy mapping between model objects and a relational database is possible with *ActiveRecord*. It promotes the use of conventions to minimize configuration (Convention over Configuration) or coding.

5.4 Infrastructure

This section describes the infrastructure with which the implementation works. However, other setups of this infrastructure are possible without major changes.

5.4.1 NSLU2

The ***NSLU2***[14] is a relatively small device with two *USB* ports and an *Ethernet* plug. It was initially developed as a *Network-Attached Storage* (*NAS*) to access *USB* mass storage devices from an *Ethernet* home network by *Linksys*. It was introduced in 2004 and discontinued in 2008 to be replaced by new products. It can be easily customized and is being used for many other applications. It is the reason why it was used to test the implementation.

The equipment has limited resources. It has an *ARM*-compatible *Intel XScale IXP420 CPU* with 32*MB* of *SDRAM*, and 8*MB* of *Flash memory*. The *CPU* was initially underclocked at 133*MHz*, but *NSLU2s* released after 2006 use the full 266*MHz* capacity. Additionally, it is equipped with a 100*Mbit/s* Ethernet network connection and two *USB 2.0* ports. See Section 2.2.1 on page 17 for more details.

5.4.2 DHCP

The *NSLU2* server has an *IP* address dynamically attributed by the *DHCP*. The computer has different *IP* addresses to simulate commands coming from different subsystems. The *Ethernet* network uses *IPv4*, but it should be possible to use *IPv6*. There is no *IP* address coded in the source code of the software, as the system relies on service discovery. See Section 4.2.1 on page 48 for more details.

The *NSLU2s* acting as gateways provide non-conflicting ranges of *IP* addresses through *DHCP* servers. It enables a bridge to obtain an *IP* address valid on the whole network in a simple and redundant manner. As long as there is at least one gateway connected to the network, *IP* addresses can be delivered to clients. Additionally, defining a non-conflicting range of *IP* addresses for a gateway to manage permits to use the simple *DHCP* redundancy mechanism. This technique is often used in an organizational environment where *DHCP* redundancy is critical.

5.4.3 Web-Service Delivery

Web services (*WS*) run on a web server. There exist many general-purpose web servers running on *Linux*. The most common one is *Apache HTTP Server* (commonly *Apache*). It offers many features and can be extended with compiled modules. It provides support for *HTTPS, Ruby, caching, content negotiation, compression*, etc.

Other general-purpose web server exist, such as *lighttpd* or *nginx*. They aimed at providing a light and fast web server. For example, they both solve the “C10k problem” which consist of handling 10 000 connections simultaneously. They offer many of the same features as *Apache*. This implementation uses ***nginx***[27] as it seemed to be the most used between the two. However, this choice is subjective and does not have a major influence on the rest of the implementation.

Additionally, another module is needed to interpret the *Ruby* web-service implementation. Again, many possibilities exist. The most common are *Mongrel*, *WEBrick*, and *Phusion Passenger*. Any of the three work perfectly to run the implementation with any of the above three general-purpose web servers.

5.5 SERVICES

This section presents the implementation of the services provided by the system. The implementation follows the design provided in Section 4.2 on page 48.

5.5.1 SIMPLE DISCOVERY PROTOCOL

The protocol follows strictly the description in Subsection 4.2.1.1 on page 48. The program implementing the service sends periodic announcement packets. The time period and the port number to send packets can be defined. The packets are *UDP multicast* packets. They contain the string: "`SDP:`", the version number of the protocol ("`1.0`"), and the *IP* address of the gateway (e.g. "`192.168.1.10`"). It also possible to send a host name, if *DNS* resolution is available on the network.

The service is implemented as a *daemon* written in *Ruby* running on gateways. It listens for incoming multicast packets on the port *2195*[1] (by default) sent by a bridge. The reply is sent on the port *2196*.

5.5.2 SELECTOR

The selector is implemented as a simple *HTTP* web service. It parses the "identifier" and "functionality" parameters provided in the request. This is done in a *RESTful* manner. It follows the logic specified in Subsection 4.2.1.3 on page 52 to find the set of devices requested. The group identifiers and functionality dependencies associated with devices are saved. When a valid set is requested, an *HTML, XML*, or *JSON* list of devices is returned to the client. In case of invalid requests, exceptions are returned as defined in Listing 4.2 on page 53.

Groups can be defined for devices with the resource "`/groups`". Afterward, groups of devices can be selected with the resource "`/selector`". It authorizes the use of logical operator (`NOT`, `AND`, `OR`). A request such as "`http://192.168.1.10/selector.xml?group=group1+AND+group2`" returns an *XML* list of devices that are in the group "group1" and "group2".

5.5.3 REST COMMUNICATION ARCHITECTURE

This part of the implementation uses intensively the *Rails* framework. It is inspired by the Python implementation that was written during the initial *HomePort* design.

[1]Port numbers 2195 and 2196 are officially unassigned by the *Internet Assigned Numbers Authority* (*IANA*) in charge of maintaining the official assignments of port numbers.

There is a resource named "`/devices`". It is associated with a model that render general data about a device (see Subsection 4.2.2.3 on page 60 for more details), except its state. Data requests are rendered by views in *HTML, XML*, and *JSON*. Requests and responses are managed by a controller. It supports requests described in Subsection 4.2.2.1 on page 53.

Two additional views were generated to enable the creation and update of device in *HTML* with a web browser. These views are simple *HTML* forms that send requests the regular *REST* manipulation interface of a gateway. For example, "`https://192.168.1.10/devices/`*`device-id`*`/edit`" provides an *HTML* form to edit the device "*device-id*".

Data exchange format

The XML data exchange format follows the description given in Subsection 4.2.2.3 on page 60. It is used as the default format to represent states of devices for the bridge. Additionally, the *JSON* format is available to facilitate *JavaScript* interactions with *AJAX*. Finally, it can be displayed in standard HTML for being used in a web browser. Internally, the representation is saved by the framework in a relational *SQL* database. In case of invalid requests, exceptions are returned as defined in Listing 4.3 on page 55.

BRIDGE

The bridge is implemented in the form of a library and a *command-line interface* (*CLI*) tool that uses the library. It is written in *Ruby*. It works by sending XML `GET`, `POST`, `PUT`, and `DELETE` requests to a gateway.

The bridge *CLI* tool follows this syntax: `http[s]://`*`gateway-address`*`|- [:`*`port`*`]` *`commands`*. Table 5.1 on the following page shows the possible *commands* and their actions. Where:

gateway-address is the address of the gateway to interact with.

- if a "-" is provided instead the address of the gateway, the automatic gateway discovery is used.

port is the port to use to connect to the server, if omitted the standard port is used.

device-id is a valid identifier for a device.

state-name is a valid name for a state.

DeviceFile.xml is a valid path to an XML file describing a device (see Subsection 4.2.2.3 on page 60 for more details).

StateFile.xml is a valid path to an XML file describing a state of a device (see Subsection 4.2.2.3 on page 60 for more details).

5.5.4 NOTIFICATION

The notification service is implemented with a daemon following the ***WS-Eventing*** *standard and is intended to run on on the gateways.* It uses the *Ruby SOAP* library to implement a *WS-Eventing*-compliant service. It provides the functionality needed to notify when a state is modified (see Subsection 4.2.2.2 on page 56 for more details).

Request	Argument 1	Argument 2	Argument 3	Action
list	-	-	-	Display of all devices with their states
list	*device-id*	-	-	Display the device with all its states
list	*device-id*	*state-name*	-	Display the state of the device
add	*DeviceFile.xml*		-	Add the device with the data in the file
add	*device-id*	*StateFile.xml*	-	Add the state to the device with the data in the file
update	*device-id*	*DeviceFile.xml*	-	Update the device with the data in the file
update	*device-id*	*state-name*	*StateFile.xml*	Update the state of the device with the data in the file
delete	*device-id*	-	-	Remove the device
delete	*device-id*	*state-name*	-	Remove the state of the device

Table 5.1: List of commands with requests, arguments, and their actions.

Ruby on Rails provides an additional abstraction level with another interface. This interface is *RESTful* and makes it easier to manage the notification. However, it simply passes requests to the *WS-Eventing SOAP daemon*.

When a change happens in the state of a device, the device communication service notifies the change to the *notification daemon*. At that point, the *daemon* notifies possible subscribers of the event-notification of the state. The notification message is sent as described during the event notification subscription. Generally, this means sending the new state to the client.

5.5.5 Hypertext Transfer Protocol Secure

The security of the web services (see Section 4.2.3 on page 62 for more details) is assured by using *HTTPS* connections between the equipment. The client has the choice to decide between regular non-secure *HTTP* connections and slower *HTTPS* connections. *nginx* supports both options in the implementation. It is possible to force the use of the *HTTPS* connections. Every equipment possesses an *SSL* certificate. A ***certificate authority*** (*CA*) was created for the system. It can generates *SSL* server certificates for gateways and *SSL* client certificates for bridges. It enables the encryption and authentication of both parties, if required.

The *CA* and the certificates were generated using *OpenSSL*[35]. It was used because it is common and efficient implementation of *SSL*. However, any other tool could be used, as it follows the *SSL* and *X509* standards. The equipment possesses a list of *CA* that they trust (see Section 2.6.2 on page 28 for more details).

5.5.6 Redundancy

Gateway

When a gateway is designated to be responsible for a bridge, it sends a *multicast* message with the *IP address* of the bridge and its own to the other gateways (*port 2188*). It is also possible to ask which gateway is responsible for a particular device by sending a *multicast* message with the *identifier* of the device (*port 2189*).

When a multicast message is sent to designate a new responsible gateway or the communication with the bridge fails, the current responsible gateway removes its internal data about devices of that bridge.

Each daemon has two mapping lists. One that matches identifiers of devices with a bridge, and another one that matches a bridge with a gateway.

Based on these lists, the gateway receiving a request relays it to the responsible gateway.

Bridge

When selecting a gateway, the bridge simply creates (`POST` requests) the devices that it controls with their states. It is the normal procedure for a bridge. Indeed, the redundancy is transparent for bridges.

The bridge periodically sends ping messages to its responsible gateway to detect eventual failures (every 3 seconds on the port 2194, by default).

5.6 Process

Environment

The development and implementation is done on an *NSLU2* with *Linux* installed on it. It is used as a server hosting the different services. Commands coming from devices trough the device layer are simulated by sending messages that devices could send. A testing program written in *Ruby* on a computer sends the simulated commands on the Ethernet network that connects the *NSLU2*. Most of the development was done locally on a virtual machine similar the *NSLU2* production environment. It enables faster development and can easily be ported to the *NSLU2* when ready for production.

Correctness testing

During the development process, the web-service were tested in 3 different manners: unit tests, functional tests, and integration tests.

The unit test verifies the model. They are the smallest testable part of the service. It is done validating individual methods. Each method has to pass a set of assertions to be validated. It controls the correctness of outputs given various inputs.

The functional test verifies the *controller* (see Section 5.3 and Figure 5.1 on pages 74–80 for more details). They verify incoming requests and responses of the rendering views. By validating the web service behavior, they verify that it follows its requirements. They test parts of the service as if it was a black box as they test it behavior, not its internal design.

The integration test verifies the integration between controllers. They test the work flow

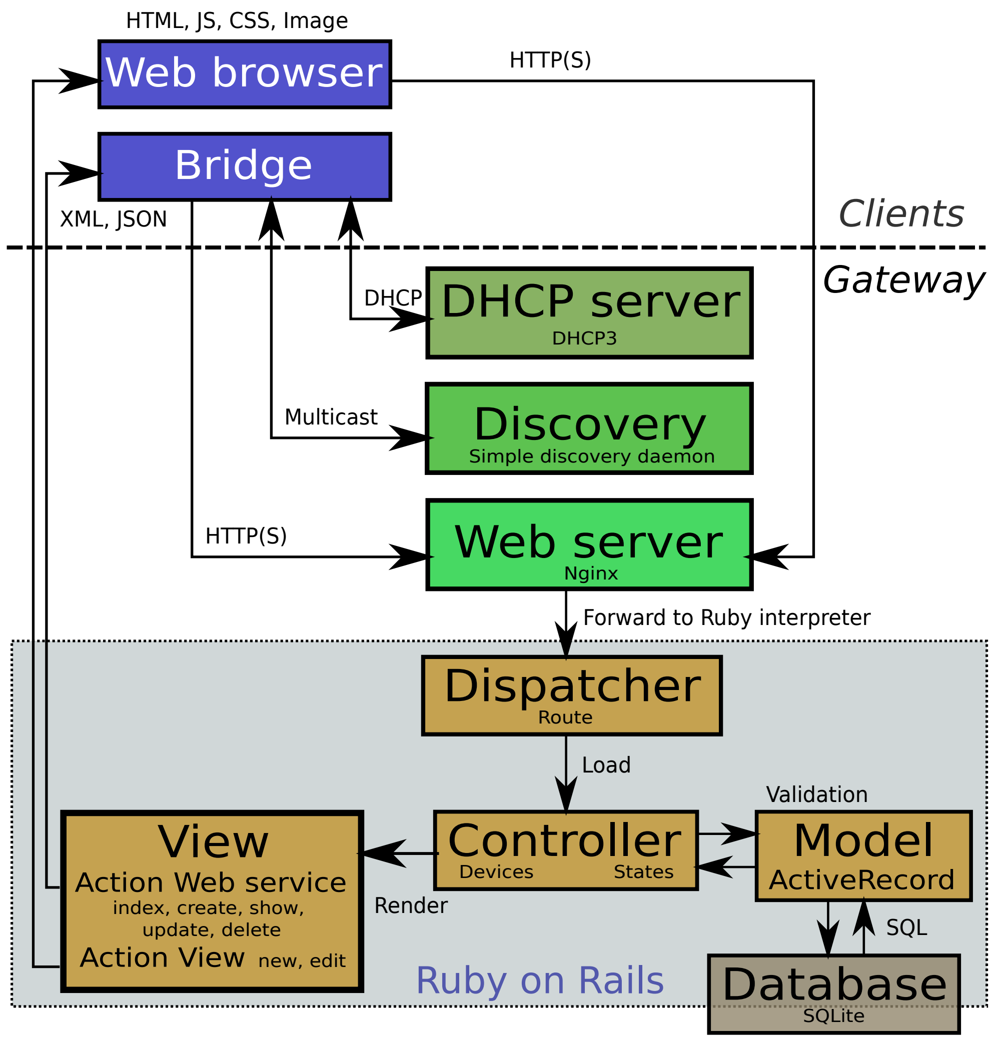


Figure 5.1: Partial summary of the implementation.

between them. For example, there are important relations between the device controller and the state controller. Therefore, it is important to test their integration. By validating groups of individual controllers of a service, they validate the service requirements as a whole.

Summary

Figure 5.1 on this page partially summarizes the implementation of the *HomePort* system. It shows the *DHCP* server, the discovery server, and the web server. It details the web-service implementation with the *Rails* framework and its different components.

CHAPTER 6

PERFORMANCE TESTING

6.1 METHODOLOGY

Many tools exist that can measure the response time of web services given different conditions. Indeed, it is interesting to know how a distributed system reacts given a certain number of concurrent connections, certain conditions, or certain requests. It validates theoretical models and emphasizes key performance elements.

Apache HTTP server benchmarking tool ("*ab*") [17] was used for measuring time responses in different scenarios. It is a simple, yet rich tool to measure and simulate connection loads. It offers rich outputs and the possibility to plot the results with other tools.

In order to have measurements as accurate as possible, they were repeated numerous times. However, many factors can influence the measured results. For example, if a connection is reset after a timeout expires (because of a high load), it greatly influence the mean. Additionally, requesting pseudo-randomly selected resources on different gateways influence the measured time because of the various caches (database, requests, interpreter, memory, ...) and the variable size of the content exchanged.

Section 6.2 on the current page presents measures of experiments designed to test the performances of services running on a single gateway. Section 6.3 on page 84 tests the performances of the discovery protocol with two gateways.

6.2 PERFORMANCE OF A SINGLE GATEWAY

The following results were calculated on a single gateway running on the *NSLU2* environment. The *NSLU2* was responsible for *50* devices each with *25* states. Concurrent requests are independent connections to the web service run in parallel. The requested state of a device is selected pseudo-randomly. A new connection is established for every request. *HTTP 1.1* features such as keep-alive, compression, or caching are disabled (see Subsection 2.3.1.2 on page 21 for more details) to test the implementation itself. They simulate connections from clients (e.g. bridges).

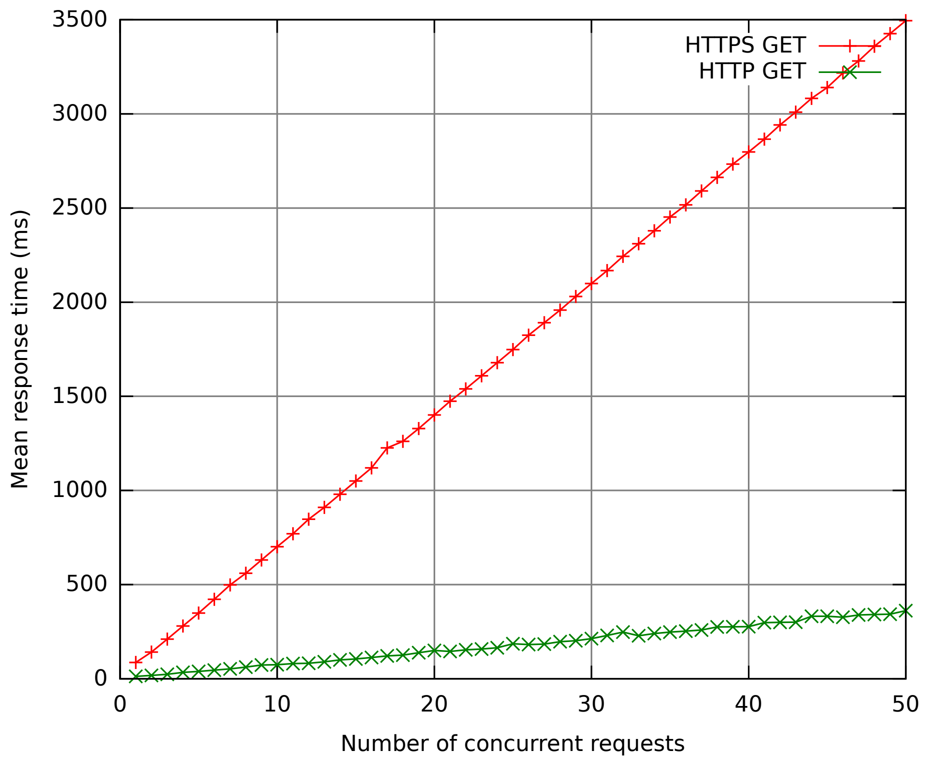


Figure 6.1: Mean response time with *HTTP* and *HTTPS* `GET` requests with concurrent requests.

6.2.1 Hypertext Transfer Protocol Secure

Figure 6.1 on this page shows the difference between the mean response time between regular *HTTP GET* requests and secure *HTTPS GET* requests. The mean was calculated on *100* measurement of concurrent *GET* request on the *XML* list of devices managed by the gateway. *The response time increases fast with concurrent HTTPS requests. Indeed, a new HTTPS session has to be established for every connection which is resource consuming. In practice, the session can be used to request more than an resource.* Maintaining the session avoids generating a new session key every time and the slow asymmetric-key operations during the session initialization from being repeated. This can be seen on Figure 6.2 on the next page. The *HTTPS* protocol uses the following parameters: `TLSv1/SSLv3, AES256-SHA, 1024, 256`. See Section 2.6.4 on page 28 for more details.

It clearly shows that *HTTPS* sessions have an high cost and limit the number of concurrent requests that can be handled by a single gateway.

6.2.2 Protocol Requests

Figure 6.3 on page 84 shows the mean response time between `GET`, `POST`, and `PUT` requests over the *HTTP* protocol. *`GET` requests are the fastest because the request does not modify the state of the device. Additionally, the request does not contain any data and can*

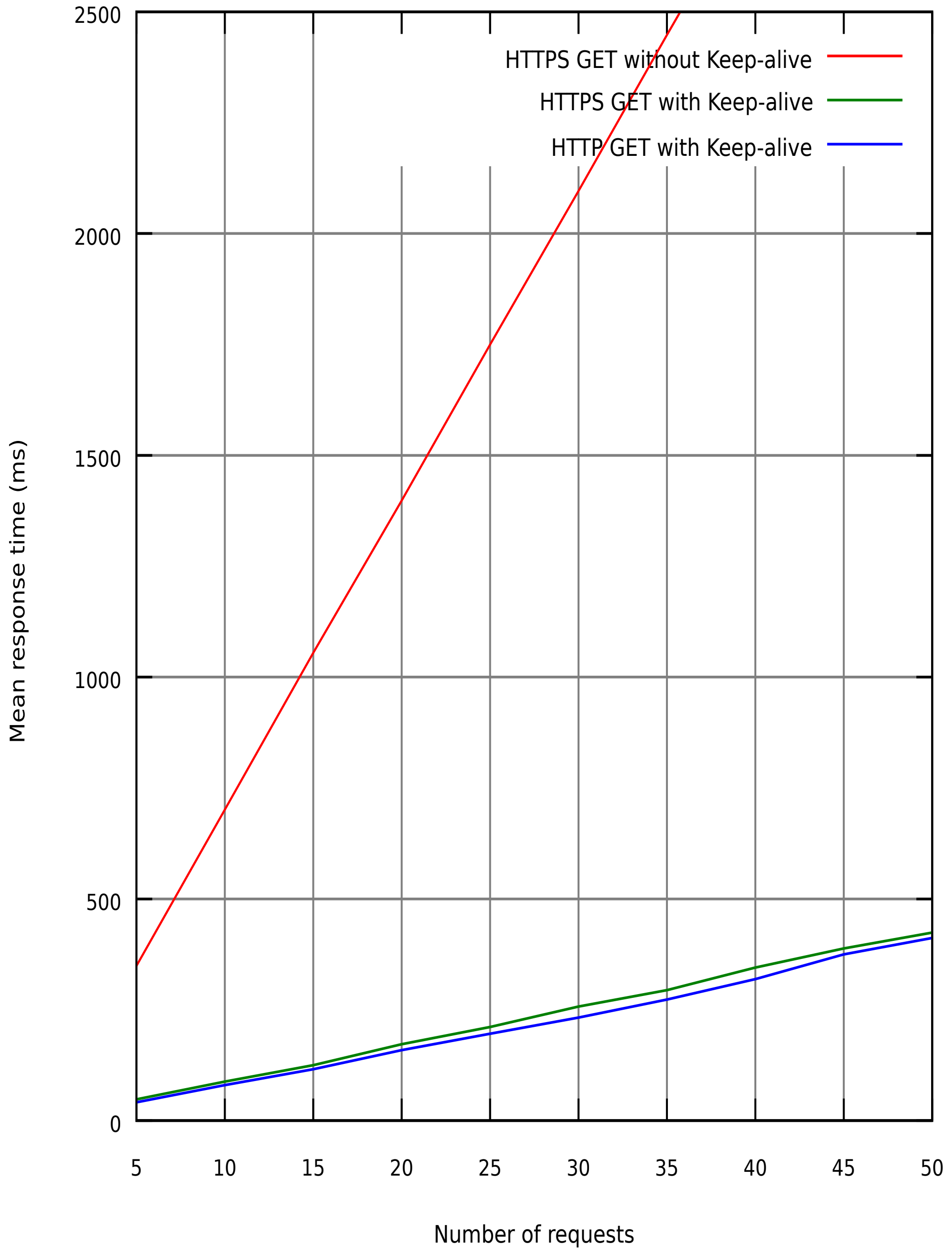


Figure 6.2: Mean response time with *HTTP* and *HTTPS* `GET` requests with and without "*Keep-alive*".

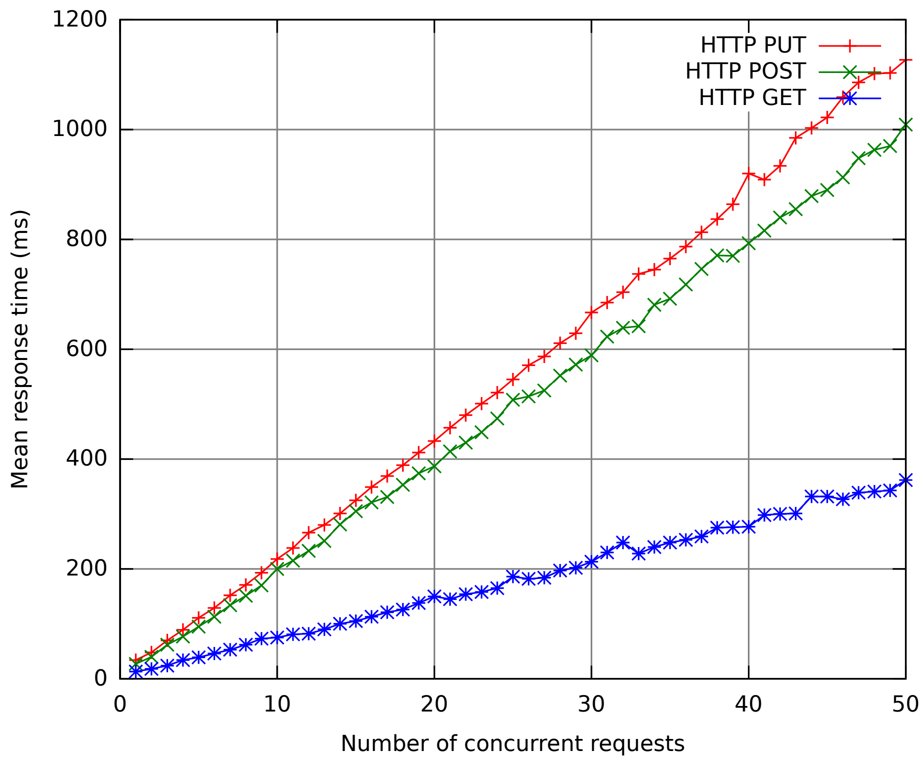


Figure 6.3: Mean response time to *HTTP* `GET`, `POST`, and `PUT` requests.

be cached. `POST` *and* `PUT` *requests have similar performances.* The slight difference can be explained by the fact that the previous state can be discarded in the case of a `POST` request, but not always in the case of `PUT` request (for which partial updates are possible). Both requests require storing a new the state of the device.

Table 6.1 on the facing page shows the minimum, mean, median, and maximum times (in *ms*) measured for different number of concurrent connections. The measurement are done on *100* repetitions. Mean and median values are closed.

Figure 6.4 on page 86 is a load test of a gateway. It shows an approximation of the number of concurrent `GET` requests that a gateway can handle with the *HTTP* and *HTTPS* protocol. The number of concurrent connection is incremented by *10* between each measurements. The mean is calculated on *50* measurements. In both cases, the response time is linear until the gateway reaches its maximum load. The maximum loads are around *260* concurrent connections with the *HTTP* protocol and *130* concurrent connections with the *HTTPS* protocol. However, a lower number of concurrent connections is needed in order to have an acceptable response time from the gateway.

6.3 Gateway-Discovery Protocol

Figure 6.5 on page 87 shows the performance of the gateway-discovery protocol under different concurrent connection loads. The measurement were realized according to the following scenario: a set of concurrent bridges sends gateway-discovery requests. Each

Concurrent requests	**Min**	**Mean**	**Median**	**Max**
HTTP GET				
1	10	13	12	79
10	10	75	48	306
20	10	150	151	515
30	10	213	222	528
40	10	277	254	804
50	10	362	391	774
60	10	437	456	1059
70	10	545	599	1256
80	38	580	621	1070
90	10	669	596	6696
100	97	734	749	3875
HTTPS GET				
1	75	87	85	174
5	106	349	348	498
10	396	701	698	962
15	411	1050	1053	1324
20	725	1401	1406	1675
30	402	2099	2106	2303
40	406	2798	2806	3070
50	423	3495	3510	3786
HTTP POST				
1	19	27	22	102
10	18	200	98	745
20	18	374	354	956
30	18	589	544	1620
40	18	793	831	1570
50	18	1009	1053	2290
HTTP PUT				
1	21	34	31	100
10	19	218	224	896
20	20	433	350	1278
30	20	667	639	1730
40	20	920	962	1818
50	20	1127	1245	2347
Concurrent requests	**Min**	**Mean**	**Median**	**Max**

Table 6.1: Main time measurements (*ms*) of requests.

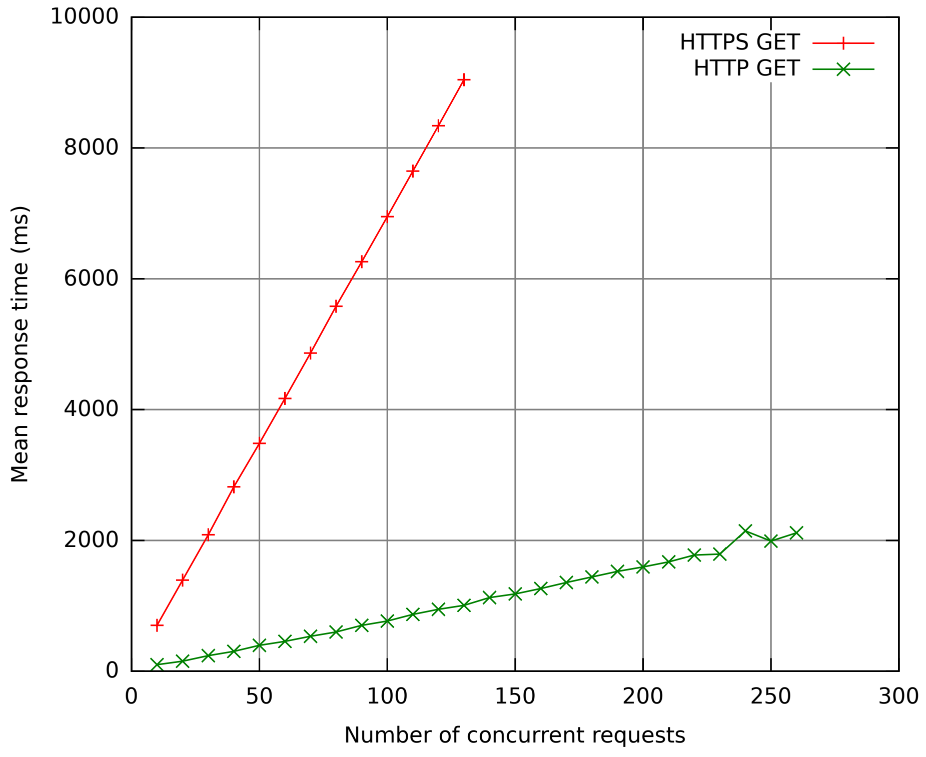


Figure 6.4: Load stress on a gateway with *HTTP* and *HTTPS* `GET` requests.

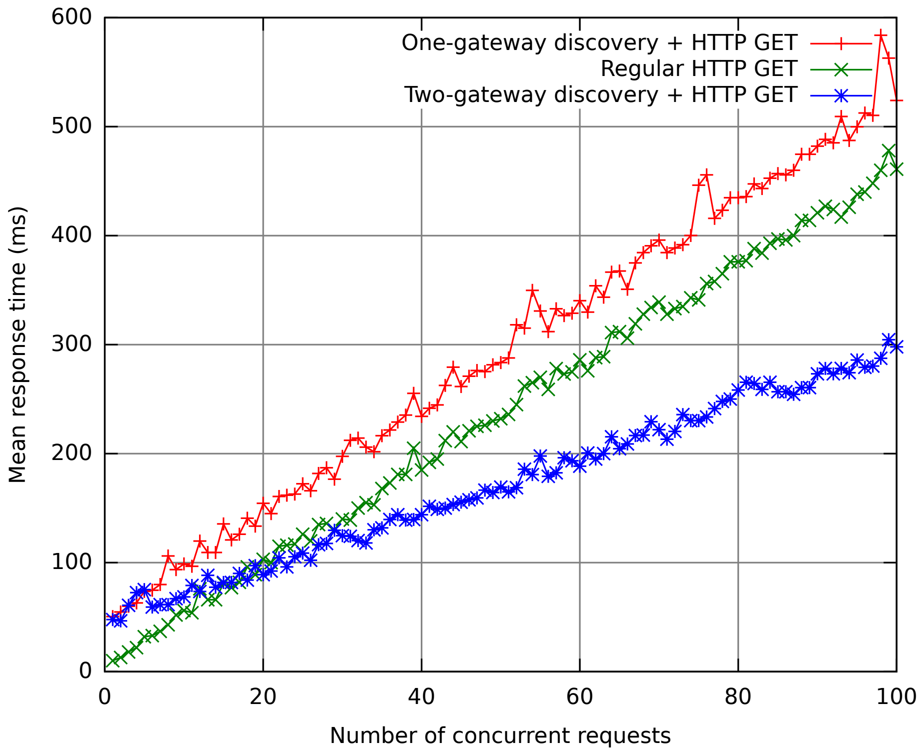


Figure 6.5: Mean response times of gateway-discovery requests followed by a *HTTP* `GET` request with one or two gateways connected, and a a *HTTP* `GET` request according to the number of concurrent connections.

individual bridge select the gateway that first responds to request. Finally, the bridge use the response of the gateway to request a resource on that gateway. The results are measured with one and two gateways connected to the network.

The discovery protocol overhead can clearly be seen when comparing a regular HTTP `GET` *with a discovery request followed by a HTTP* `GET`*. The experience proves that the discovery protocol balances fairly well the charge between the two gateways.* This can be seen when comparing mean response times between one and two gateways. It is logical consequence of the fact that when one gateway is loaded, it answers more slowly to discovery requests. Therefore, the charge goes to the other gateway.

The results are calculated with *20* measurements.

6.4 Notification Protocol

Figure 6.6 on the next page shows the time it takes for clients to modify states of devices (with a *HTTP* `PUT` request) and the other clients interested on modified states to be notified. The measurements are done with one and two gateways in the system. Both measurements are quite linear.

When a random request to update a state arrives to a gateway, it might need to relay it to the other gateway. Indeed, if the gateway is not responsible for that device, it passes

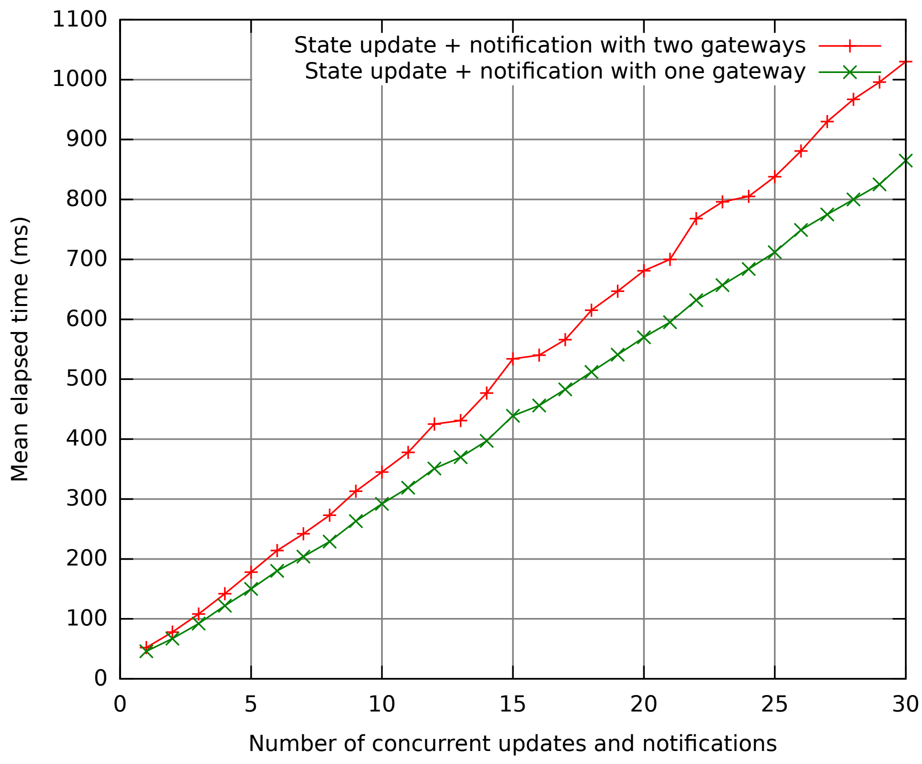


Figure 6.6: Mean times of concurrent updates of states and their notifications with one or two gateways.

the request to the responsible gateway (see Subsection 4.2.4.1 on page 64 for more details). This is why measurements with two gateways is more chaotic than with only one.

The reaction time is quite slow in both cases which shows that the implementation could be improved. However, the progression is linear in both case which is a good case for the scalability of the system (see Section 2.4.2 on page 23 for more details). This means that by tuning the implementation or adding more powerful gateways the performance can be improved.

The results are calculated with *100* measurements.

CHAPTER 7

CONCLUSION

This last chapter sums up the various main points of the work, as well as the elements for future work.

SUMMARY

The present work started by presenting the current technologies available for home automation systems. It presented network types used in modern home. Afterward, many standards related to web services were described. Indeed, the design of the original *HomePort* system, also presented, and many other modern domotic systems, heavily rely on these Internet technologies. This is an essential part to understand already existing systems and their possibilities. From this part, it is clear that already existing web-service standards can be applied to the field of domotics. In particular, much research was made about discovery, event notification, security, scalability, and reliability.

Afterwards, special care was taken to define sets of requirements for the system. It is mostly based on review of the domotic literature and discussions on the subject. It defines goals to achieve, commonly found features, and constraints of the system.

The third part of the work analyzes possible solutions based on the goals, requirements, constraints, and existing technologies. It clearly defines two protocols to enable automatic discovery of components of the system. The first protocol is simple and lightweight. It uses the possibility to broadcast packets on an *Ethernet* network in order to resolve *IP* addresses of components of the system. The second protocol takes advantages of the rich and numerous features of *WS-Discovery*. It requires more resources to use it, but offers more functionalities. Both protocols assume that components have an *IP* address on the network. *IP* addresses can be attributed with various other protocols presented.

Commands available in the system are clearly defined. They are *REST-style* methods largely based on the initial *HomePort* design. The definition of these methods with *WSDL*, combined with *WS-Discovery*, enables dynamic use of the provided commands. Properties of this architecture are expressed and analyzed. Additionally, there is a new service that enables to select sets of devices based on their functionalities or group identifiers. It facilitates the management of large or complex domotic systems.

A new event-notification feature is introduced. It enables more possibilities than the initial simple HomePort notification mechanism. It uses the principles of the *WS-Eventing* standard. Parts that are not specified in this standard were implemented such as the *SOAP* notification. This protocol enables new functionalities in the system such as selective notifications. However, it is still possible to use the original HomePort notification mechanism.

The security perimeter defines the security responsibilities of the system. It shows aspects to be protected by the system and by subsystems. This part also uses security standards to protect the different links between device components.

The redundancy and scalability of the architecture is studied.

Finally, the implementation of this new system is presented. It is mainly implemented using *Ruby* and it relies on existing libraries for web services. It is inspired from the initial *HomePort* piece of software that was written in *Python*.

Table 7.1 on the facing page summarizes contributions and their origin to the best of the author's knowledge.

FUTURE WORK

Many new features and functionalities were introduced in the present work. It would be interesting to analyze how they **integrate with real vendor subsystems**. Different practical aspects when it comes to integrate and manufacture the system should also be considered. Extensive and large-scale testing is needed.

The **mobility of devices** (both inside and outside the house) was not considered at all in the original design of the HomePort system, nor in this work. However, it should be facilitated by automatic discovery and configuration features. Advanced features might be interesting for these particular devices, such as having different behavior depending on their localization in the system. This would require further work and new features.

In parallel, a lot of work is done at the **composition layer** of the system. Efficient and practical solutions have to be developed to combine the different subsystem devices.

Formal analysis with verification tools of the protocols between equipment would ensure the safety of the architecture.

Finally, for the system to be ever used by companies, a final set of choices has to be defined in a **standard**. In order to do so, meetings and alliances with companies are necessary. Indeed alternatives are numerous and conventions are needed in order to interoperate.

There is a great interest from vendors for open home-automation solutions. In this context, the *HomePort system* has a real potential.

Name	**Origin**
Analysis of the different technological aspects of HomePort	Author
Define requirements for goals	HomePort
Define requirements for features	Author (partially based on HomePort and research)
Define requirements for constraints	HomePort (mostly)
Layered architecture	HomePort
Service oriented approach	HomePort
Automatic discovery and configuration	Author
Simple discovery protocol (design, implementation, and testing)	Author
WS-Discovery (analysis and use)	Author
Devices Profile for Web Services (analysis and use)	Author
Simple HomePort notification ("LISTEN")	HomePort
WS-Event (analysis, use, and implementation)	Author
REST architecture	HomePort
WSDL of services	Author
Multi formats data representation (analysis and use)	HomePort
XML format definition	Author
Multi formats data representation through content negotiation (HTML, text, XML, and JSON) (implementation and testing)	Author
Group and dependency identifiers ("selector")	Author
Security perimeter (analysis)	Author
HTTPS (TLS) to authenticate servers and clients (analysis, use, implementation, and testing)	Author
Availability (analysis)	Author
Multiple gateways (analysis, implementation, and testing)	Author (suggested by HomePort)
Synchronization of gateways (analysis, implementation, and testing)	Author
Scalability of requests from clients to gateways	Author (based on HomePort design)
Implementation design on NSLU2	Author (inspired from HomePort implementation)
Implementation of services on NSLU2	Author
Implementation testing	Author
Web interface	Author
Command-line interface	Author
Performance testing of different requests with different loads	Author

Table 7.1: Summary of the contributions and their origin.

Bibliography

[1] 37Signals. Ruby on rails. `http://rubyonrails.org/`, May 2010.

[2] HomePlug Powerline Alliance. Homeplug av white paper. `http://www.homeplug.org/products/whitepapers/HPAV-White-Paper_050818.pdf`, Dec. 2005.

[3] Wi-Fi Alliance. Wi-fi certified makes it wi-fi: An overview of the wi-fi alliance approach to certification. `http://www.wi-fi.org/files/WFA_Certification_Overview_WP_en.pdf`, Sept. 2006.

[4] Z-Wave Alliance. Z-wave: The new standard in wireless remote control. `http://www.z-wave.com/modules/AboutZ-Wave/`, Oct. 2009.

[5] ZigBee Alliance. Zigbee white papers. `http://www.zigbee.org/LearnMore/WhitePapers/tabid/257/Default.aspx`, Oct. 2009.

[6] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 1 edition, January 2001.

[7] Universal Powerline Association. Upa digital home specification v1.0. `http://www.upaplc.org/`, Feb. 2006.

[8] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical report, 1993.

[9] Changseok Bae, Jinho Yoo, Kyuchang Kang, Yoonsik Choe, and Jeunwoo Lee. Home server for home digital service environments. *Consumer Electronics, IEEE Transactions on*, 49(4):1129–1135, Nov. 2003.

[10] Gordon Bell and Jim Gemmell. A call for the home media network. *Commun. ACM*, 45(7):71–75, 2002.

[11] Peter Bergstrom, Kevin Driscoll, and John Kimball. Making home automation communications secure. *Computer*, 34(10):50–56, 2001.

[12] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-soa home control gateway. In *Services Computing, 2006. SCC '06. IEEE International Conference on*, pages 463–470, Sept. 2006.

[13] Jeppe Brønsted, Per Printz Madsen, Arne Skou, and Rune Torbesen. The homeport system. In *2010 IEEE Consumer Communications and Networking Conference (CCNC)*, Jan. 2010.

[14] Inc. Cisco Systems. Nslu2. `http://homesupport.cisco.com/en-us/wireless/lbc/NSLU2`, May 2010.

[15] Ruby Community. Ruby. `http://www.ruby-lang.org/`, May 2010.

[16] Roy Thomas Fielding. Chapter 5: Representational state transfer (rest). `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`, 2000.

[17] The Apache Software Foundation. Apache http server benchmarking tool. `http://httpd.apache.org/docs/2.3/programs/ab.html`, May 2010.

[18] ITU. X.200 : Information technology - open systems interconnection - basic reference model: The basic model. `http://www.itu.int/rec/T-REC-X.200-199407-I/en`, Jul. 1994.

[19] F. Jammes, A. Mensch, and H. Smit. Service-oriented device communications using the devices profile for web services. In *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, volume 1, pages 947–955, May 2007.

[20] F. Jammes and H. Smit. Service-oriented architectures for devices - the sirena view. In *Industrial Informatics, 2005. INDIN '05. 2005 3rd IEEE International Conference on*, pages 140–147, Aug. 2005.

[21] W. Kastner, G. Neugschwandtner, S. Soucek, and H.M. Newmann. Communication systems for building automation and control. *Proceedings of the IEEE*, 93(6):1178–1203, June 2005.

[22] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[23] Robert E. McGrath. Discovery and its discontents: Discovery protocols for ubiquitous computing. Technical report, Champaign, IL, USA, 2000.

[24] B.A. Miller, T. Nixon, C. Tai, and M.D. Wood. Home networking with universal plug and play. *Communications Magazine, IEEE*, 39(12):104–109, Dec 2001.

[25] Almut Herzog Nahid and Nahid Shahmehri. Towards secure e-services: Risk analysis of a home automation service. In *In Proceedings of the 6 th Nordic Workshop on Secure IT Systems (NordSec*, pages 18–26, 2001.

[26] M. Nakamura, Y. Fukuoka, H. Igaki, and K.-i. Matsumoto. Implementing multivendor home network system with vendor-neutral services and dynamic service binding. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 275–282, July 2008.

[27] nginx. nginx. `http://nginx.org/`, May 2010.

[28] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld, and Mathilde Pignol. Generating remote control interfaces for complex appliances. pages 161–170. ACM Press, 2002.

[29] OASIS. Devices profile for web services version 1.1. `http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html`, Jul. 2009.

[30] Melissa J. Perenson of PC World. Superspeed usb 3.0: More details emerge. `http://www.pcworld.com/article/156494/superspeed_usb_30_more_details_emerge.html`, Jan. 2009.

[31] Linux Online. The linux home page. `http://www.linux.org/`, May 2010.

[32] Paolo Pellegrino, Dario Bonino, and Fulvio Corno. Domotic house gateway. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1915–1920, New York, NY, USA, 2006. ACM.

[33] J. Plonnigs, M. Neugebauer, and K. Kabitzsch. A traffic model for networked devices in the building automation. In *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, pages 137–145, 2004.

[34] Universal Plug and Play Forum. Upnp ressources. `http://www.upnp.org/resources/default.asp`, Sept. 2009.

[35] The OpenSSL Project. Openssl. `http://www.openssl.org/`, May 2010.

[36] B. Rose. Home networks: a standards perspective. *Communications Magazine, IEEE*, 39(12):78–85, Dec 2001.

[37] U. Saif, D. Gordon, and D. Greaves. Internet access to a home area network. *Internet Computing, IEEE*, 5(1):54–63, Jan/Feb 2001.

[38] Bluetooth Special Interest Group (SIG). Bluetooth. `https://www.bluetooth.org/apps/content/`, Sept. 2009.

[39] IEEE Computer Society. 802.11 part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. `http://standards.ieee.org/getieee802/download/802.11-2007.pdf`, June 2007.

[40] The Internet Society. Hypertext transfer protocol – http/1.1 (rfc 2616). `http://tools.ietf.org/html/rfc2616`, June 1999.

[41] The Internet Society. Http over tls (rfc 2818). `http://tools.ietf.org/html/rfc2818`, May 2000.

[42] The Internet Society. Security architecture for the internet protocol (rfc 4301). `http://tools.ietf.org/html/rfc4301`, Dec. 2005.

[43] The Internet Society. The transport layer security (tls) protocol version 1.2 (rfc 5246). `http://tools.ietf.org/html/rfc5246`, Aug. 2008.

[44] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[45] USB Implementers Forum (USB-IF). Universal serial bus. `http://www.usb.org/`, Sept. 2009.

[46] W3C. (extensible markup language) xml-signature syntax and processing (rfc 3275). http://tools.ietf.org/html/rfc3275, March 2002.

[47] W3C. Web services architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, Feb. 2004.

[48] W3C. Web services eventing (ws-eventing). http://www.w3.org/Submission/2006/SUBM-WS-Eventing-20060315/, March 2006.

[49] W3C. Soap version 1.2 part 1: Messaging framework (second edition). http://www.w3.org/TR/2007/REC-soap12-part1-20070427/, April 2007.

[50] W3C. Web services description language (wsdl) version 2.0 part 2: Adjuncts. http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626/, June 2007.

[51] Thomas Lauterbach Wolfgang Hoeg. *Digital audio broadcasting: principles and applications of digital radio*. John Wiley and Sons, 2003.

[52] Ben Yan, Masahide Nakamura, Lydie du Bousquet, and Ken ichi Matsumoto. Validating safety for the integrated services of the home network system using jml. *Journal of Information Processing*, 16:38–49, 2008.

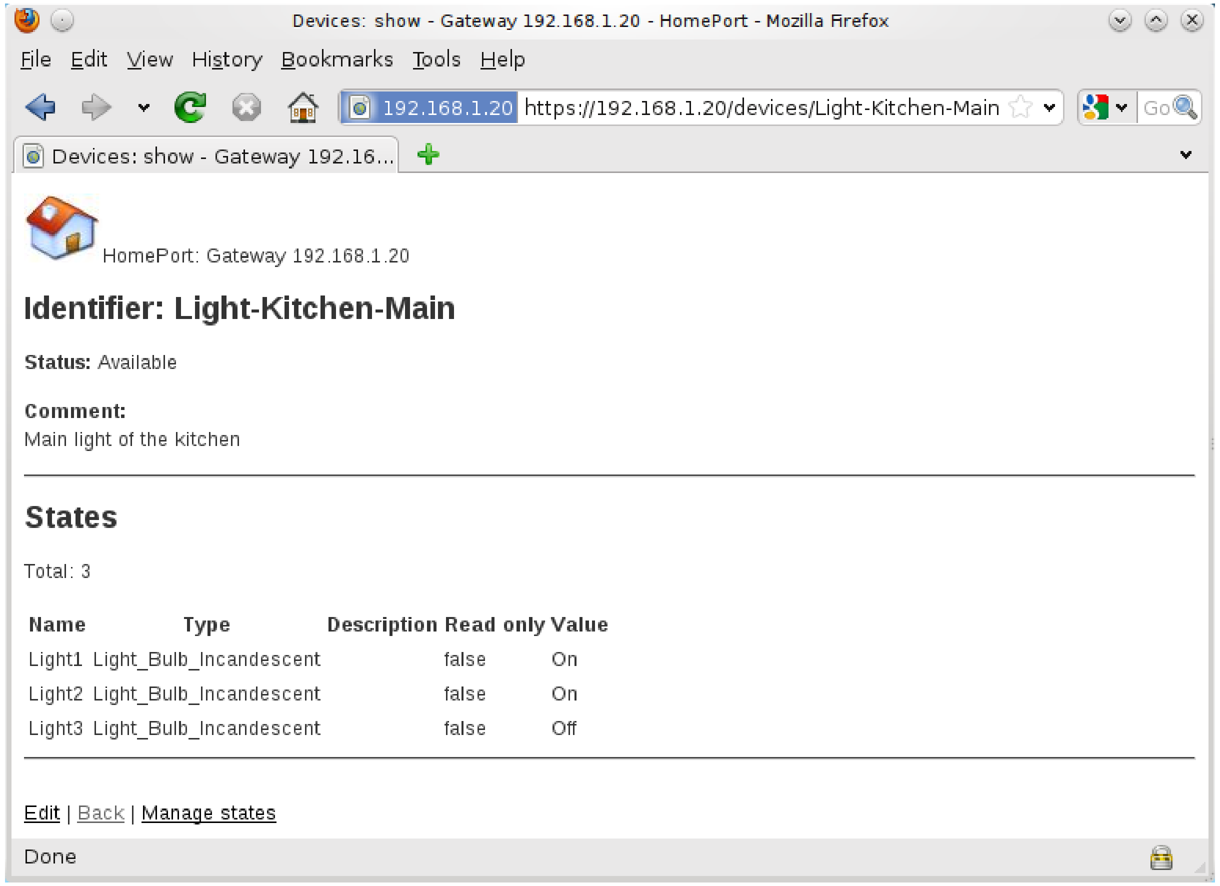
Index

APPENDIX A

INTERFACES

A.1 WEB INTERFACE OF GATEWAYS

This section contains screen shots with a few of the views in the web-browser interface of the implementation.


Devices: show - Gateway 192.168.1.20 - HomePort - Mozilla Firefox
File Edit View History Bookmarks Tools Help
192.168.1.20 https://192.168.1.20/devices/Light-Kitchen-Main
Devices: show - Gateway 192.16...

HomePort: Gateway 192.168.1.20

Identifier: Light-Kitchen-Main

Status: Available

Comment:
Main light of the kitchen

States

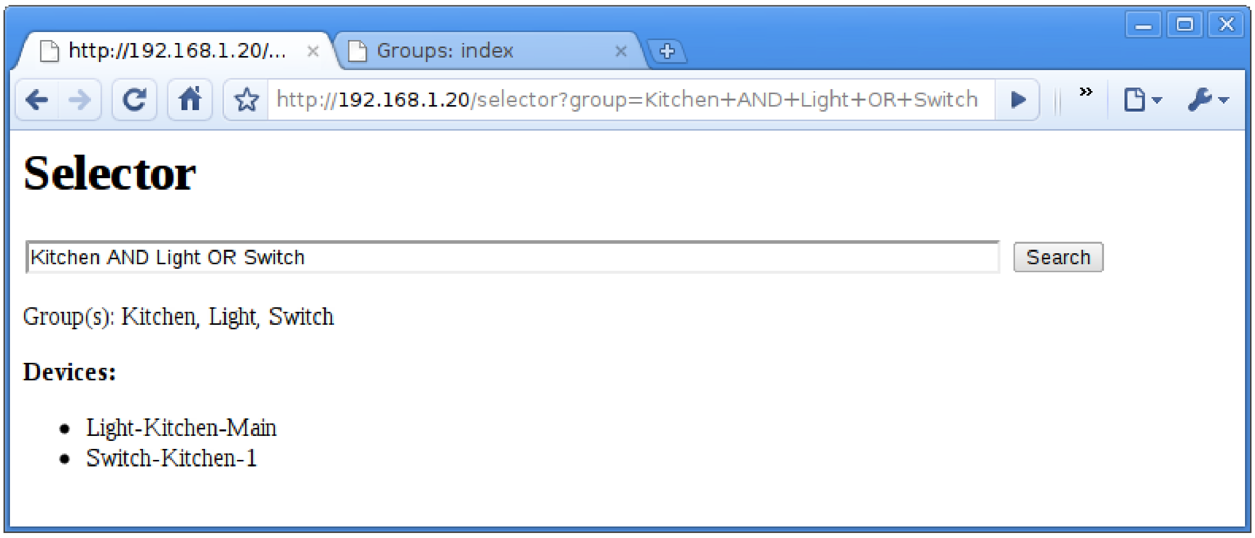
Total: 3

Name	Type	Description	Read only	Value
Light1	Light_Bulb_Incandescent		false	On
Light2	Light_Bulb_Incandescent		false	On
Light3	Light_Bulb_Incandescent		false	Off

Edit | Back | Manage states

Done

Figure A.1: Display data about a device.


http://192.168.1.20/...
Groups: index
http://192.168.1.20/selector?group=Kitchen+AND+Light+OR+Switch

Selector

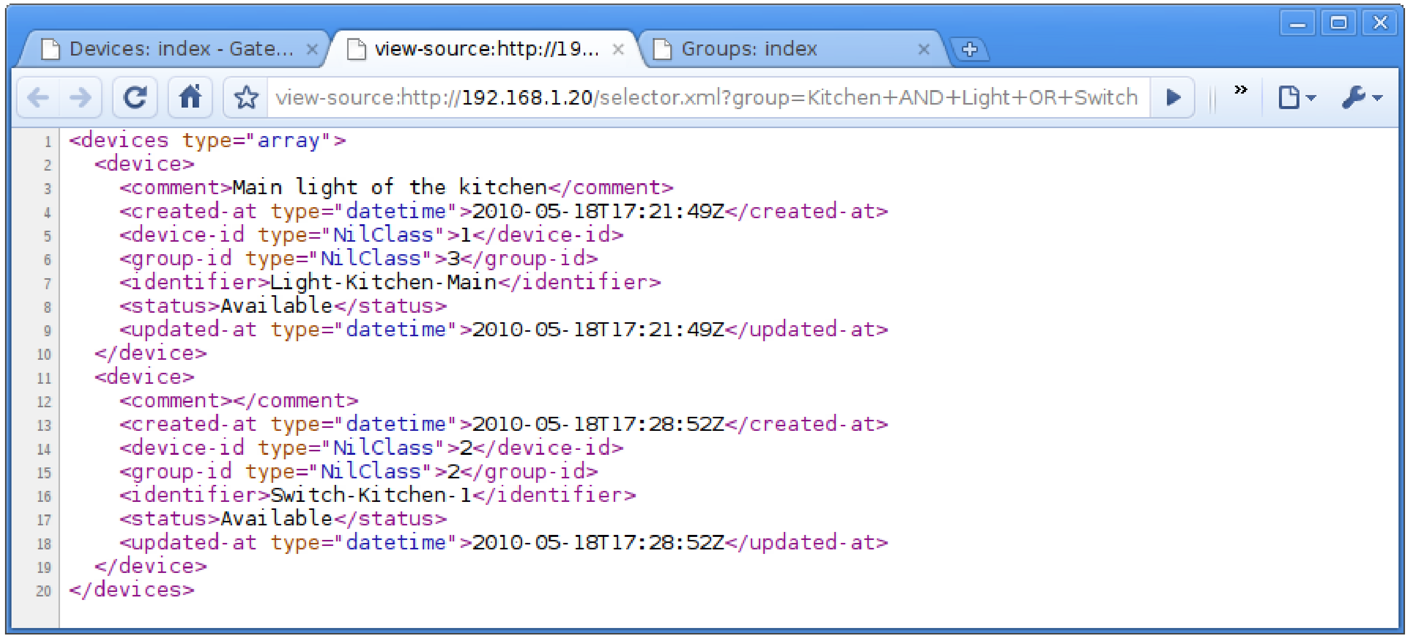
Kitchen AND Light OR Switch

Group(s): Kitchen, Light, Switch

Devices:

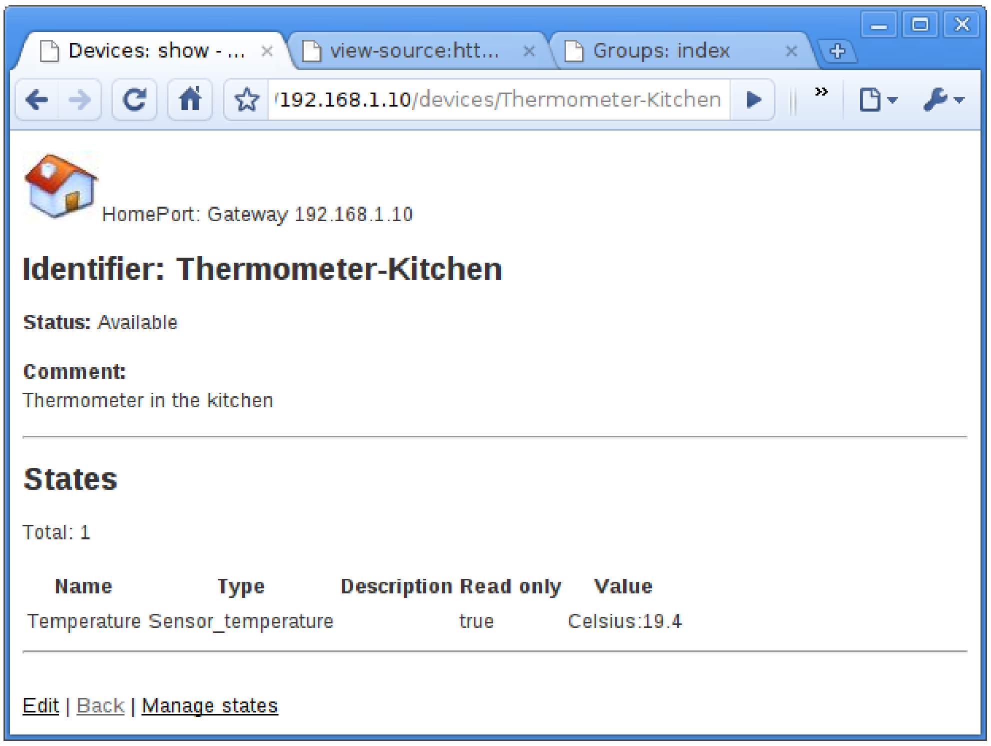
- Light-Kitchen-Main
- Switch-Kitchen-1

Figure A.2: Display device search with the selector.


Devices: index - Gate...
view-source:http://19...
Groups: index
view-source:http://192.168.1.20/selector.xml?group=Kitchen+AND+Light+OR+Switch

```
<devices type="array">
  <device>
    <comment>Main light of the kitchen</comment>
    <created-at type="datetime">2010-05-18T17:21:49Z</created-at>
    <device-id type="NilClass">1</device-id>
    <group-id type="NilClass">3</group-id>
    <identifier>Light-Kitchen-Main</identifier>
    <status>Available</status>
    <updated-at type="datetime">2010-05-18T17:21:49Z</updated-at>
  </device>
  <device>
    <comment></comment>
    <created-at type="datetime">2010-05-18T17:28:52Z</created-at>
    <device-id type="NilClass">2</device-id>
    <group-id type="NilClass">2</group-id>
    <identifier>Switch-Kitchen-1</identifier>
    <status>Available</status>
    <updated-at type="datetime">2010-05-18T17:28:52Z</updated-at>
  </device>
</devices>
```

Figure A.3: Display XML results of the device selector.



HomePort: Gateway 192.168.1.10

Identifier: Thermometer-Kitchen

Status: Available

Comment:
Thermometer in the kitchen

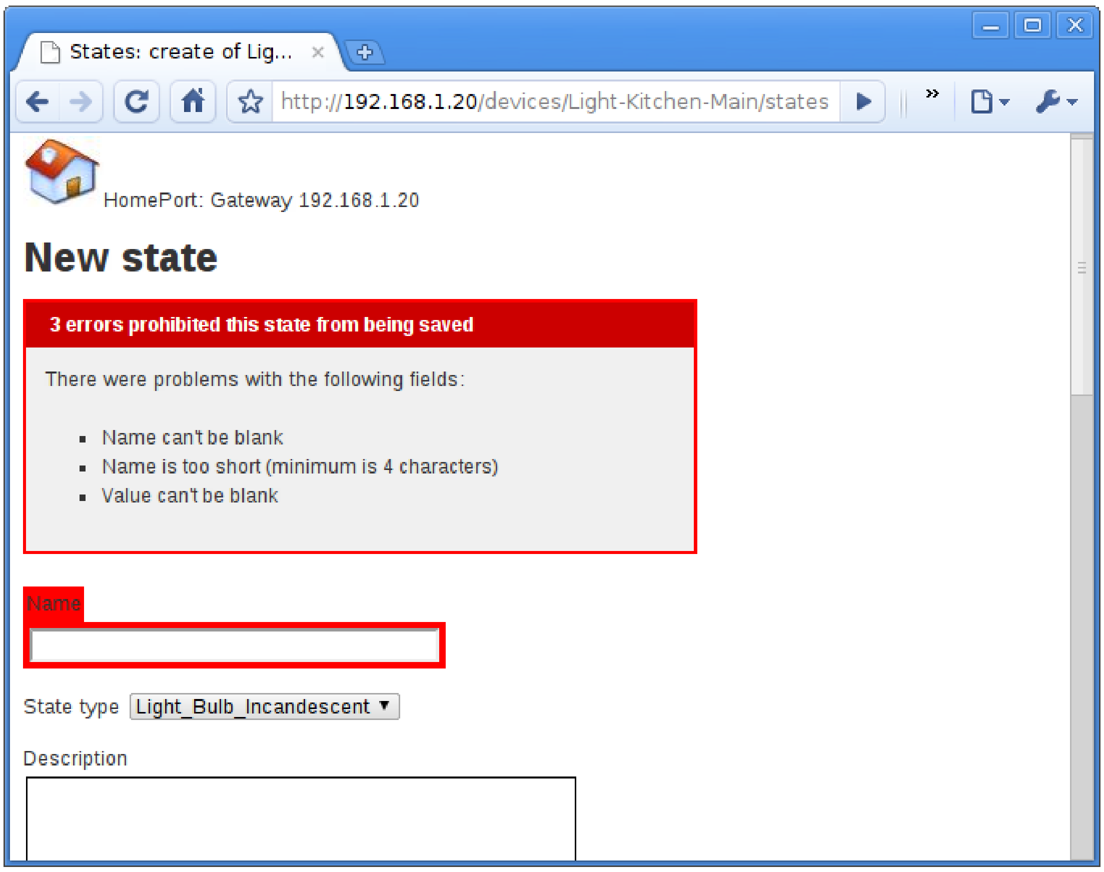
States

Total: 1

Name	Type	Description	Read only	Value
Temperature	Sensor_temperature		true	Celsius:19.4

Edit | Back | Manage states

Figure A.4: Display the state of the *Thermometer-Kitchen* device.



HomePort: Gateway 192.168.1.20

New state

3 errors prohibited this state from being saved

There were problems with the following fields:

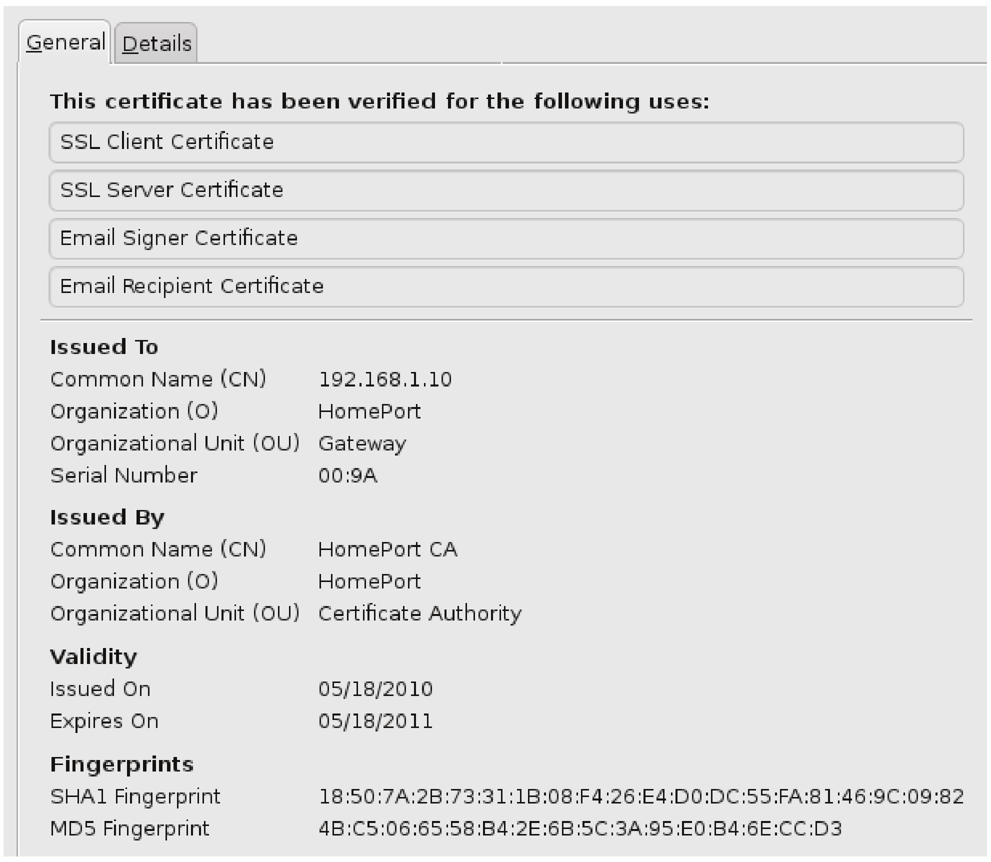
- Name can't be blank
- Name is too short (minimum is 4 characters)
- Value can't be blank

Name

State type Light_Bulb_Incandescent

Description

Figure A.5: Display error message when trying to create an invalid new state.



This certificate has been verified for the following uses:

SSL Client Certificate

SSL Server Certificate

Email Signer Certificate

Email Recipient Certificate

Issued To

Common Name (CN)	192.168.1.10
Organization (O)	HomePort
Organizational Unit (OU)	Gateway
Serial Number	00:9A

Issued By

Common Name (CN)	HomePort CA
Organization (O)	HomePort
Organizational Unit (OU)	Certificate Authority

Validity

Issued On	05/18/2010
Expires On	05/18/2011

Fingerprints

SHA1 Fingerprint	18:50:7A:2B:73:31:1B:08:F4:26:E4:D0:DC:55:FA:81:46:9C:09:82
MD5 Fingerprint	4B:C5:06:65:58:B4:2E:6B:5C:3A:95:E0:B4:6E:CC:D3

Figure A.6: User view of the security mechanisms with a standard web-browser interface.

A.2 Command-Line Interface For Clients

This section provides listing of different commands and their outputs of the command-line interface tool. The following listings start with the command (“*$*”) executed in a terminal (see Section 5.5.3 on page 77 for more details), followed by the output from the tool.

Listing A.1: List of states of a device (with gateway discovery enabled)

```
$ device – list Thermometer–Kitchen
Automatically discovering gateway...
Discovered gateway: http://192.168.1.20

Response:


<?xml version="1.0" encoding="UTF–8"?>
<device>
  <comment>Thermometer in the kitchen</comment>
  <created–at type="datetime">2010–05–18T17:29:54Z</created–at>
  <identifier>Thermometer–Kitchen</identifier>
  <status>Available</status>
  <updated–at type="datetime">2010–05–18T17:29:54Z</updated–at>
  <states type="array">
    <state>
      <name>Temperature</name>
      <state–type>Sensor_temperature</state–type>
    </state>
  </states>
</device>
```

Listing A.2: List of devices (with gateway discovery enabled)

```
$ device – list
Automatically discovering gateway...
Discovered gateway: http://192.168.1.10

Response:


<?xml version="1.0" encoding="UTF–8"?>
<devices type="array">
  <device>
    <comment>Main light of the kitchen</comment>
    <created–at type="datetime">2010–05–18T17:21:49Z</created–at>
    <identifier>Light–Kitchen–Main</identifier>
    <status>Available</status>
    <updated–at type="datetime">2010–05–18T17:21:49Z</updated–at>
  </device>
  <device>
    <comment></comment>
    <created-at type="datetime">2010-05-18T17:28:52Z</created-at>
    <identifier>Switch-Kitchen-1</identifier>
    <status>Available</status>
    <updated-at type="datetime">2010-05-18T17:28:52Z</updated-at>
  </device>
  <device>
    <comment>Thermometer in the kitchen</comment>
    <created-at type="datetime">2010-05-18T17:29:54Z</created-at>
    <identifier>Thermometer-Kitchen</identifier>
    <status>Available</status>
    <updated-at type="datetime">2010-05-18T17:29:54Z</updated-at>
  </device>
</devices>
```

Listing A.3: Read the state of a device (with gateway discovery enabled)

```
$ device - list Thermometer-Kitchen Temperature
Automatically discovering gateway...
Discovered gateway: http://192.168.1.10

Response:


<?xml version="1.0" encoding="UTF-8"?>
<state>
  <created-at type="datetime">2010-05-18T17:32:18Z</created-at>
  <description></description>
  <name>Temperature</name>
  <read-only-state type="boolean">true</read-only-state>
  <updated-at type="datetime">2010-05-18T17:32:18Z</updated-at>
  <value type="celsius">19.4</value>
  <state-type>Sensor_temperature</state-type>
</state>
```

Listing A.4: Update the state of a device (with direct connection)

```
$ device https://192.168.1.20 update Switch-Kitchen-1 Position
    NewPosition.xml
/devices/Switch-Kitchen-1/states/Position.xml
#<Net::HTTPOK:0x7f9ab780db68>
```

Listing A.5: Delete the state of a device (with direct connection)

```
$ device https://192.168.1.20 delete Switch-Kitchen-1 Position
/devices/Switch-Kitchen-1/states/Position.xml
#<Net::HTTPOK:0x7fa3b29fbbd8>
```

Listing A.6: Errors for requesting a unknown state of a device

```
$ device https://192.168.1.20 list Switch-Kitchen-1 XXX
<?xml version="1.0" encoding="UTF-8"?>
<errors>
  <error>Name does not exist</error>
</errors>
```

APPENDIX B

RÉSUMÉ

More and more devices of our daily life are computer based. This trend can be seen in many different areas of home equipment. Most communication systems are based on electronic components that run specifically designed programs.

Overall, electronic and computer-based devices are more reliable than purely mechanical devices, based on equivalent functional-requirement complexity. Indeed, electronic components are not subject to the same physical deterioration as mechanical ones. Additionally, they offer unique and powerful possibilities. For example, it is possible to modify the behavior of a device after it was build, by changing its *firmware* or *software*. It is also easier for users to customize their devices to fit their needs. Contents of diverse kind can be copied and exchanged in an efficient and cheap manner. Finally, these devices appear smarter to the users.

The normal development direction has turned from isolated and purpose-specific devices to collaborative and multi-purpose devices. Nowadays, most cellphones cannot only make phone calls, but also surf the *Internet*, take pictures, interact with other devices (through *Bluetooth*), run user applications, etc. They interact with the cellphone network, the *Internet*, various wireless networks, and home computers. The same is true for computers, gaming consoles, and media players.

All these devices need to interact with each other through various networks. Some networks are specific for a segment of devices and others are general for heterogeneous devices.

This trend to make daily-life devices digital happened in a disorganized manner. Many technologies were developed in parallel. Furthermore, many categories of devices have their own standards and means of communication. In addition, devices are diverse, and they have different constraints and resources. In order to obtain truly collaborating devices, a system that can handle various technologies is needed.

Such a system would help equipment to communicate, and act in an efficient and smarter manner. Moreover, it has to be flexible enough to adapt to numerous devices and their needs. The system can help to meet the new challenges that are low-energy consumption, easy configuration, security and affordable price for such a complex system.

The objective of this work is to provide an home-automation system. The system is based on the design of the *HomePort* system. The *HomePort* system was designed to offer a distributed communicating architecture that provides an added value. It uses a layered and service-oriented architecture.

The current work presents the context in which home-automation systems are used. It describes and compares home-automation related technologies in general, and in the specific case where a service-oriented approach is used. Moreover, different technologies to enhance the original HomePort system are also presented.

As a starting point, the requirements of the system are defined from the existing home-automation literature. Thereafter, various solutions to problems that were left for future work in the initial HomePort architecture are described.

Solutions include the automatic discovery and configuration of the components of the system. It applies the *Devices Profile for Web Services* to the system. It clearly defines two protocols to enable automatic discovery of components of the system. The first protocol is simple and lightweight. It uses the possibility to broadcast packets on an *Ethernet* network in order to resolve *IP* addresses of components of the system. The second protocol takes advantages of the rich and numerous features of *WS-Discovery*. It requires more resources to use it, but offers more functionalities. Both protocols assume that components have an *IP* address on the network. *IP* addresses can be attributed with various other protocols presented. In order to reach acceptable user requirements, it extends and defines new functionalities, such as the group and dependency identifiers, and dynamic component discovery.

Commands available in the system are clearly defined. They are *REST-style* methods largely based on the initial *HomePort* design. The definition of these methods with *WSDL*, combined with *WS-Discovery*, enables dynamic use of the provided commands. Properties of this architecture are expressed and analyzed. Additionally, there is a new service that enables to select sets of devices based on their functionalities or group identifiers. It facilitates the management of large or complex domotic systems.

A new event-notification feature is introduced. It enables more possibilities than the initial simple HomePort notification mechanism. It uses the principles of the *WS-Eventing* standard. Parts that are not specified in this standard were implemented such as the *SOAP* notification. This protocol enables new functionalities in the system such as selective notifications. However, it is still possible to use the original HomePort notification mechanism.

Additionally, it includes the description of different security mechanisms, at various system levels, compatible with the rest of the system. It starts by defining a security perimeter of the system. It explains the extend to which the system can protect the user, given the constraints. It proposes different possible solutions to the threats.

Moreover, these solutions are implemented to be used and integrated with the *HomePort* system. The implementation context is also described. Choices made for the implementation are described and possible alternative solutions are presented.

Finally, the performance of the implementation is measured and analyzed. It underlines limitations of this implementation on the available hardware equipment. Additionally, the consequences of the implementation based on the design are emphasized.

APPENDIX C

CD-ROM

Here is a CD-Rom containing the implementation source code, tools, XML files, raw-data measurements, and interface screen shots.