Outsourcery
Simply extending home automation using plugins

by

T.L.M. Brands
L.J. Helsloot
G.M.A. Wassenaar

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Friday June 26, 2015 at 08:45 AM.

Supervisor: Dr. ir. J. A. Pouwelse, TU Delft
Thesis committee: Dr. ir. F. F. J. Hermans, TU Delft
M. A. Larson, TU Delft
E. Marges, HomeWizard B.V.

An electronic version of this thesis is available at http://repository.tudelft.nl/.
This document describes Outsourcery, a plugin system developed for the Noordwijk-based home automation company HomeWizard. It is part of a final bachelor project concluding the Bachelor of Science program in Computer Science at the Delft University of Technology. The report gives a complete overview of the software development process and the delivered product.

The goal of this document is to describe the development phases, give an overview of the performed work and explain choices made during the design of the product. Furthermore, this document contains recommendations for HomeWizard employees to successfully integrate and use the system in their home automation hub.

Some minor parts of this document contain confidential information and should be redacted before publication. These parts are marked with [REDACTED]. We have done our best to minimize the amount of redacted text, and to make the rest of the document understandable without the confidential information.

T.L.M. Brands
L.J. Helsloot
G.M.A. Wassenaar
Delft, June 2015
HomeWizard is developing a low-budget home automation hub with multi-vendor compliance. The hub must be able to control hundreds of different home automation devices from dozens of vendors. Since a growing number of home automation devices can be controlled using TCP/IP, HomeWizard is interested in a system that properly controls devices using TCP/IP.

The large number of networking standards used in home automation poses a problem for a low-budget hub, as many standards require costly subscriptions or certified chipsets. Connecting these networks to TCP/IP networks via gateways, however, allows HomeWizard to provide a home automation solution that is affordable, yet compatible with a large range of devices.

At the moment, adding device compatibility to the HomeWizard hub is a labour intensive task for the HomeWizard developers. Therefore, HomeWizard wants a software solution that enables hardware manufacturers to make their own TCP/IP-enabled devices compatible with the HomeWizard hub through plugins. Adding support for a new device should not require changes to the firmware of the HomeWizard hub, nor to the connected device. Furthermore, since some hardware manufacturers may not have programming experience, developing a plugin should not require advanced programming skills.

The project objective is to develop a prototype of a plugin system for HomeWizard. This plugin system should allow hardware manufacturers to add compatibility for TCP/IP-enabled devices to the HomeWizard hub. People with little programming experience should be able to develop a plugin for a simple device within one day.

Based on research into the technical background of the problem, including different device types, existing standards and required functionality, three successive prototypes are described. These prototypes, developed according to the primary design goals flexibility and simplicity, correspond to the three scrum sprints used to iterate on the product design.

The first prototype, named the iKettle implementation, is focused around a fully functional plugin implementation for the iKettle, an exotic device ignoring all standards. This prototype enables the use of a simple plugin to control a single IP/TCP-enabled device, providing a basic structure for the following prototypes. The prototype developed during the second two-week scrum cycle, called ‘the Pixar Prototype’, focuses on security by separating plugins in different processes. Moreover, communication with devices via gateways is implemented. The final developed prototype, named Outsourcery, is the product as released to HomeWizard. It focuses on delivering the ‘complete package’, including an acceptance test, a simple demo interface and a dedicated developer tutorial.

Outsourcery is a successful solution to the posed problem, providing a flexible and simple plugin system for the HomeWizard home automation hub. It allows manufacturers of TCP/IP-enabled home automation devices to easily make their devices compatible with the HomeWizard hub, thereby providing customers with a versatile yet affordable home automation solution. The indicated future work and the given recommendations will help turn Outsourcery into a stable and mature product, allowing HomeWizard to get to its fullest potential.
We would like to thank Dr.ir. J.A. Pouwelse from the Parallel and Distributed Systems department of the TU Delft, for his guidance and counsel throughout the project. We thank E. Marges from HomeWizard B.V., for his hospitality and the pleasant collaboration. Additionally, we are grateful for the provided equipment and facilities. We thank Dr.ir. F.F.J. Hermans and M.A. Larson for supervising the project.

Special thanks go out to the entire HomeWizard development team for providing us with the necessary tools and support, and Bas in particular for helping us perform the user acceptance test.
Contents

Preface i
Summary ii
Acknowledgements iii
1 Introduction 1
  2 Problem Description 3
    2.1 Limitations .............................................. 4
    2.2 Device types ........................................... 5
3 Methodology 6
4 Goals & requirements 7
  4.1 Design goals ........................................... 7
  4.2 Requirements ........................................... 9
5 Technical background 10
  5.1 Existing modular platforms for home automation ................................ 10
  5.2 Network techniques for home automation ........................................ 11
  5.3 Used programming languages .................................... 13
  5.4 Device control steps ........................................ 14
6 The iKettle implementation 16
  6.1 The iKettle Plugin ........................................ 16
  6.2 Core architecture .......................................... 20
  6.3 Dependency Management ..................................... 21
  6.4 Documentation and Testing .................................. 21
  6.5 Encountered problems ...................................... 21
  6.6 Next sprint ................................................ 22
7 The Pixar Prototype 23
  7.1 Separation of processes ..................................... 24
  7.2 The HomeWizard gateway plugin ................................. 25
  7.3 Important changes and reuse of code ............................. 28
  7.4 Documentation and Testing .................................. 30
  7.5 Encountered problems ...................................... 30
  7.6 SIG feedback ............................................. 30
  7.7 Next sprint ................................................ 30
8 Outsourcing Outsourcery 31
  8.1 Outsourcery ............................................... 31
  8.2 User acceptance test ....................................... 33
  8.3 SIG feedback ............................................. 34
9 Evaluation and recommendations 35
  9.1 Evaluation of the product .................................... 35
  9.2 Future work ............................................... 37
  9.3 Recommendations ........................................... 38
HomeWizard B.V. is a company that aims to connect existing consumer products into a seamless and affordable home automation experience. The company develops a central home automation hub, capable of controlling 100+ different devices from 20+ vendors via WiFi and the 433 MHz and 868 MHz radio frequency (RF) wavelengths. The aim of HomeWizard is to provide a cost-effective home automation solution with multi-vendor compliance [1]. HomeWizard develops only the central home automation hub, a device to control heating and a device to monitor energy usage (figure 1.1). Many more related products marketed by other companies can be used in concert with the central HomeWizard hub, including, but not limited to, the Philips Hue, IP cameras and smoke detectors.

Figure 1.1: Devices created by HomeWizard are the central hub, the EnergyLink for energy usage monitoring and the HeatLink for controlling heating.

HomeWizard provides smartphone and web applications to allow users to control devices connected to the central home hub. The application can be used to control devices directly, e.g. turning on a light, or to create ‘tasks’, one or more actions bound to triggers. Triggers can be based on device properties or the current time, whereas actions can control devices or send notifications.

A detailed description of the problem is provided in chapter 2. The methodology used during this project to develop a software solution for this problem is described in chapter 3. Design goals and requirements guiding the development process are listed in chapter 4, and chapter 5 contains research into the technical background of the problem. The final software solution is implemented during three
implementation sprints. During the first sprint, detailed in chapter 6, a single device plugin is implemented. The second sprint, explained in chapter 7, supports gateway devices and adds process separation. The last sprint focused on delivering the 'complete package', including an acceptance test, a simple demo interface and a dedicated developer tutorial, as can be read in chapter 8. After the implementation sprints, an evaluation of the product is done to ascertain that the delivered product meets the expectations of HomeWizard. Furthermore, an overview of future work and recommendations for HomeWizard are given. This can be found in chapter 9. Finally, a conclusion is provided in chapter 10.
Problem Description

HomeWizard is developing a low-budget home automation hub with multi-vendor compliance. The hub must be able to control hundreds of different home automation devices from dozens of vendors, with support for new devices being added regularly.

An explosion of new networking standards has resulted from recent developments in the Internet of Things. New standards on data link, network and application layers have been introduced by various researchers and companies. Attempts to create home networks have led to even more standards and protocols, including ZigBee, Z-Wave and KNX [2, 3]. This has caused the home automation market to become heavily fragmented, with different vendors using different standards and open or proprietary protocols.

The large number of standards poses a problem to a low-cost home automation hub as developed by HomeWizard. Because many standards require costly subscriptions or certified chipsets, HomeWizard cannot offer a cost-effective solution that supports all available standards with dedicated hardware.

Controlling devices using TCP/IP

A growing number of home automation devices can be controlled using TCP/IP. Moreover, many other home networks, such as ZigBee, can be connected to TCP/IP networks via gateways. HomeWizard developers believe that by supporting TCP/IP devices, including gateways to reach other types of networks, they can develop a low-budget device with multi-standard compliance. Figure 2.1 shows an example of how devices using different communication protocols can be connected to the HomeWizard hub via TCP/IP.

Communication with TCP/IP-enabled devices, however, proceeds using many different protocols. While some devices use common communication standards, such as HTTP, others use proprietary TCP- or UDP-based protocols. Moreover, each device has its own set of control instructions and responses.

Adding device compatibility to the HomeWizard hub is currently very labour-intensive. Adding support for a device in the HomeWizard hub requires HomeWizard developers to manually change the hub’s firmware to include device-specific control instructions and protocols. To reduce the time required to add device support, while increasing the number of supported devices, HomeWizard wants to enable device manufacturers to connect their device to the HomeWizard hub themselves.

Software solution: device compatibility simplified

HomeWizard needs a software solution that enables hardware manufacturers to make their devices compatible with the HomeWizard hub. Adding support for a new device should not require changes to the firmware of the HomeWizard hub, nor to the connected device. To achieve this, the HomeWizard hub must be capable of using control instructions specified by the manufacturer.

Plugins should be used to define control instructions for devices. By dynamically loading and executing these plugins in userspace, support for TCP/IP devices can be added to the HomeWizard hub at runtime without requiring changes to the hub’s firmware.

Plugin development should be simple. Since some hardware manufacturers may not have programming experience, developing a plugin should not require advanced programming skills. People with little programming experience should be able to write a plugin for a simple device within one day.
2.1. Limitations

HomeWizard platform  The created prototype must run on the HomeWizard hub.

---

Project objective

The objective of this project is to develop a prototype of a plugin system for HomeWizard. This plugin system should allow hardware manufacturers to add compatibility for TCP/IP-enabled devices to the HomeWizard hub. People with little programming experience should be able to develop a plugin for a simple device within one day.

With manufacturers being able to connect their devices to the hub, the HomeWizard developers no longer have to implement control instructions for each device themselves. This speeds up and simplifies the process of connecting devices for both the producer of the hub and the manufacturer of the home automation device. This reduces the high cost associated with home automation, thus making it more accessible.

Exposing device capabilities

For devices to be integrated into the trigger and action model utilized by HomeWizard, plugins must expose device capabilities in terms of data that can be triggered on and possible actions to control devices. Although integration with existing HomeWizard software is outside the scope of this project, the system must provide a means for such integration to be implemented in the future. Interpretation of exposed capabilities, supplying parameters to actions and creation of a graphical interface for the end user are not within the scope of this project.

---

Figure 2.1: An example of a HomeWizard home automation network. The HomeWizard hub can communicate directly with IP devices, but can only reach others, such as ZigBee and Z-Wave devices, via gateways. These gateways, in turn, are reached through TCP/IP, be it using a well-known protocol such as HTTP or a proprietary UDP or TCP protocol.
8 Weeks time  The prototype must be developed within 8 weeks to adhere to the timeline of the project.

No budget  As no budget is available for this project, only free software and standards must be used.

2.2. Device types
The method of using gateway devices to connect the HomeWizard hub to other types of networks, as depicted in figure 2.1, poses difficulties to implementing uniform communication with TCP/IP devices. Rather than treating every device connected via TCP/IP as a single device, some devices may act as a gateway to a number of other devices and should be treated as multiple devices. This calls for a distinction between different types of devices.

Single device  A single device which can be controlled using a direct connection via IP. The single device possesses a WiFi connection and control instructions are sent directly to the device. The Smarter™ wifikettle, also known as iKettle\textsuperscript{1}, is such a single device. The IP-enabled kettle can be controlled directly via a TCP connection, over which commands and status messages are exchanged.

Dedicated gateway  A dedicated gateway is a gateway to a network of devices with a fixed instruction set. The gateway can only be used to communicate with specific devices. Control instructions, sent from the central hub over TCP/IP, are addressed only to the gateway, which in turn controls the devices connected to the dedicated gateway. The Philips Hue bridge, as shown in figure 2.2a is such a dedicated gateway, which can control home lighting with predefined presets using an HTTP REST API.

Generic gateway  A generic gateway is a device that only passes on instructions to another network. The gateway receives messages via TCP/IP, translates those messages to another standard and sends the messages on to the other network. Contrary to a dedicated gateway, a generic gateway generally has no knowledge of how the devices connected to the network should be controlled, and thus requires the central hub to provide the control instructions for those devices, as well as control instructions for the generic gateway itself. The Digi XBee to IP gateway\textsuperscript{2}, as shown in figure 2.2b, is such a generic gateway that can connect XBee networks to IP networks.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2-2.png}
\caption{Two types of gateway devices.}
\end{figure}

\textsuperscript{1}iKettle Wifi Kettle - http://smarter.am\textsuperscript{2}Digi XBee Gateway - http://www.digi.com/products/wireless-routers-gateways/gateways/xbee-gateway
Methodology

The project lasts from April 20 until June 26, 2015 (10 weeks). Because the students are at the client’s office for three days a week, consults with the client can easily be arranged. A final prototype will be completed on the 19th of June. The following working agreements are applied during the project.

Scrum
The agile project management method scrum is applied using this project. Short sprints of 2 weeks are practiced for quick prototyping to deliver multiple prototypes. The product and sprint backlogs are maintained using a Trello project board.

Definition of done
After each scrum sprint, a fully functional prototype is delivered. A prototype is considered “done” when it is fully functional, the corresponding documentation is complete, and the code is well-structured and commented. Furthermore, the prototype must have passed all test suites.

Testing methods
The code is accompanied by test suites, which utilize unit testing. Every prototype undergoes a regression test. Manual testing and integration testing is also used.

Version control
For version control, a BitBucket git repository is used. Bitbucket is already used by Homewizard’s developers, making it easy to share the project sources with the HomeWizard development team. New features are developed on separate branches. Branches are integrated using pull requests. When pull requests are approved by all team members, they are merged into the development branch. Working prototypes are placed in the master branch.

Documentation
Every sprint will come with its own documentation chapter (see chapters 6, 7 and 8), explaining the design of the resulting prototypes. In these chapters, the choice for the implemented features will be explained and encountered problems will be discussed.

Source code is documented using documentation comments, which are extracted to API documentation in HTML format.

1Trello project boards - https://trello.com
Goals & requirements

Design goals and requirements are used to guide the development of the software system. Along with the project objective, the design goals and requirements can be regarded as the success criteria of this project. The design goals, explained in section 4.1, describe which properties are desirable in the software system. The design goals differ from the requirements, listed in section 4.2, because the design goals describe conflicting properties that are desirable, but not always feasible, whereas the requirements describe features that must be present in the software system.

4.1. Design goals

While developing a software solution for the problem described in chapter 2, a number of design goals should be kept in mind. These design goals describe the most important aspects, as derived from the problem description. The design goals are grouped into five categories, each describing a different aspect. The categories are listed in order of importance, i.e. flexibility is the most important design goal, followed by security, then performance and lastly maintainability.

4.1.1. Flexibility

Any system that can cooperate with a wide variety of existing as well as future devices must be flexible enough to adapt to all these devices. For HomeWizard, this means that it should not only be possible to use existing communication protocols, but also that it should be possible to add new protocols that might be created in the future to the system. To give a more thorough definition of the required flexibility, it has been split into two different design goals, both of which are described below. A third design goal, simplicity, describes flexibility in terms of usage of the system.

Extensible

The system should be capable of supporting proprietary and little-used protocols used by manufacturers of devices, as well as future advances in communication protocols. The future, however, is notoriously hard to predict. Therefore, the system must be extensible. Breivold et al. [4] define extensibility as "The capability of the software system to enable the implementation of extensions to expand or enhance the system with new capabilities and features with minimal impact to the existing system". For HomeWizard, this means that it must be possible to add new communication protocols to the system without having to change the system's firmware.

Modular

Naab and Stammel [5] claim the key principle behind realizing flexibility at architectural level is modularity. Modular software increases flexibility by allowing one module to be written with little knowledge of another module, as well as allowing modules to be replaced without changing the rest of the system [6]. Modularity allows the system to use plugins as building blocks, using one or more plugins that are necessary to control a device without having to change the rest of the system.
4.1.2. Simplicity
Plugin development should be simple. While creating complex plugins for sophisticated devices should be possible, simple devices should not require advanced knowledge of programming or complex configuration. People with little programming experience should be able to develop a plugin for a simple device within one day.

4.1.3. Security
One of the barriers preventing adoption of home automation systems in households is the lack of security offered by existing systems. Especially high-concern devices such as door locks and cameras raise security concerns [7]. Although the approach of connecting existing devices to the HomeWizard hub cannot increase the level of security between the hub and connected devices (because the firmware of those devices cannot be changed), the system should allow reaching the maximum level of security offered by each device. Furthermore, the HomeWizard hub should be protected against malicious and badly-written plugins. Plugins should thus not be able to read or modify information unrelated to the device they are controlling.

4.1.4. Performance
The hardware limitations described in section 2.1 impose some constraints on the computational resources available to the system. To provide a fluent user experience, the system should use no more resources than are available. Moreover, the dynamic nature of a plugin-based system calls for the core of the system, i.e. the parts of the system that are not plugins, to use no more resources than required, leaving more resources available for plugins. The performance of the system is split into four design goals listed below.

Lightweight
The system should require little resources. Memory usage, CPU usage and lengthy input/output operations should be minimized. Moreover, as plugins can be the cause of performance bugs [8], plugins should not inherently require many resources.

Responsive
The system should process control instructions within an acceptable amount of time. Although the response time of a request depends on many variables outside the control of the system, such as other software running on the HomeWizard hub, performance of plugins, and the response time of external devices, the system itself should not introduce a delay noticeable to the user.

Stable
Plugins should not interfere with each other or with other systems running on the HomeWizard hub. It should not be possible for plugins to read or modify any information other than that related to the devices controlled by the plugin. Moreover, a fault in a plugin should not cause instability to other systems on the HomeWizard hub.

Scalable
The system should not directly limit users in the number of devices they can connect, nor should device manufacturers be limited in the number of plugins that can be loaded. The limiting factor should be the capabilities of the underlying hardware, so the number of devices that can be connected increases when the hardware capabilities of the HomeWizard hub increase.

4.1.5. Maintainability
Although the system is designed with future developments in mind, accurately predicting such developments is hard, if not impossible. Despite best efforts to create a flexible system, the system will most likely need to be changed in the future. Therefore, the system must be setup in such a way that modifications can be made easily.

Modular
Apart from benefiting flexibility, modularity is also an important aspect of maintainability. Modularity allows changing individual system components without requiring extensive adaptations in other
components. Modularity improves testability as well as extensibility of the system, thus increasing maintainability.

**Testable**
Non-trivial code should be tested with unit tests to ensure quality. Regression tests should be used to uncover new bugs in existing components. The overall system should be tested using integration tests.

**Documented**
All exposed source code components should be documented using source code comments. Source code comments should be extracted to create a structured and readable API reference, e.g. in HTML format. Moreover, explanatory comments should be added to non-trivial source code used internally.

### 4.2. Requirements
The design goals described in section 4.1 form the basis for a list of requirements that clarify the required functionality of the software solution. The requirements are ordered by their importance, requirement 1 being most important. Requirement 1, describing the functionality of the prototype, is not based on any design goal but reflects HomeWizard’s expectations.

<table>
<thead>
<tr>
<th>#</th>
<th>Requirement</th>
<th>Design goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The prototype must be capable of controlling at least one switch via TCP/IP.</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>It must be possible to implement the logic required to control devices over TCP/IP using plugins.</td>
<td>Flexibility</td>
</tr>
<tr>
<td>3</td>
<td>The system must expose the capabilities of device plugins, consisting of data that can be triggered on and actions to control devices.</td>
<td>Flexibility</td>
</tr>
<tr>
<td>4</td>
<td>It must be possible to add and remove plugins while the system is running.</td>
<td>Flexibility</td>
</tr>
<tr>
<td>5</td>
<td>The system must run on a Linux-based operating system.</td>
<td>Performance</td>
</tr>
<tr>
<td>6</td>
<td>The system must run on hardware equivalent with that of a Raspberry Pi.</td>
<td>Performance</td>
</tr>
<tr>
<td>7</td>
<td>It should be possible to implement new application layer protocols using plugins, without having to change the system’s firmware.</td>
<td>Flexibility</td>
</tr>
<tr>
<td>8</td>
<td>The system should allow reaching the maximum level of security offered by each device.</td>
<td>Security</td>
</tr>
<tr>
<td>9</td>
<td>The system should provide barriers to limit interference between plugins.</td>
<td>Security</td>
</tr>
<tr>
<td>10</td>
<td>Each module and each public function in the system should be documented</td>
<td>Maintainability</td>
</tr>
<tr>
<td>11</td>
<td>At least 85% of the code should be covered with unit tests.</td>
<td>Maintainability</td>
</tr>
<tr>
<td>12</td>
<td>After development of the system, other engineers should be able to understand and maintain the code.</td>
<td>Maintainability</td>
</tr>
<tr>
<td>13</td>
<td>The architecture of the system should adhere to Software Engineering design principles, as described in Dr. A. Bacchelli’s Software Engineering Methods course.</td>
<td>Maintainability</td>
</tr>
</tbody>
</table>
Technical background

Technical background research is conducted to give an overview of available solutions, network techniques and suitable programming languages. Section 5.1 describes existing modular platforms for home automation. In section 5.2 detailed research into network techniques is described. An evaluation of potential programming languages is given in section 5.3. This chapter concludes with an enumeration of possible required device steps in section 5.4.

5.1. Existing modular platforms for home automation

HomeWizard is not the first to attempt creating a plugin-based or otherwise modular platform for home automation. However, despite the widespread use of plugin systems to create extensible software systems, no standardized solutions to create plugin systems seem to exist. Most approaches require a plugin to either expose one or more ‘entry points’ with predetermined names or allow plugins to ‘hook’ functionality to certain events. Moreover, applications generally provide functions that can be called from a plugin to interact with the application.

While system languages such as C were historically widely used to create plugins, such as with the Netscape NPAPI [9], modern plugin systems often use scripting languages such as Javascript, as employed by Mozilla’s Addon SDK [10], or Lua, as used in World of Warcraft [11]. The lack of ready-to-use generic plugin frameworks requires a specialized plugin solution to be created for the system.

Research on existing products can provide interesting information for developing a new plugin system. Therefore, five existing products that attempt to create such a platform or are otherwise relevant are described below.

**CEBus/Internet gateway**

Desbonnet and Corcoran have presented a gateway that can connect a consumer electronics bus (CEBus) network with the internet. This gateway, programmed in Java, enables controlling devices on the CEBus network from an internet application [12]. It is equipped with a graphical user interface that lists all available devices. When a device is selected to be accessed, a user interface is downloaded from a url [13]. While this gateway is not a solution to HomeWizard’s problem, it shows that it is feasible to load a module at runtime to enable device control. Loading user interface modules at runtime is similar to loading plugins to control devices.

**Contiki**

Contiki is a low-weight, open source operating system created to run on IoT devices. One of its distinguishing features is its ability to dynamically load and replace services at runtime. This enables constrained devices to have versatile functionality despite having limited available memory. Communication is implemented as a service, allowing communication protocols to be replaced at runtime. This gives constrained devices the possibility to make use of various communication protocols with low memory costs [14]. Contiki has full IPv6 support and has implementations of 6LoWPAN, RPL and CoAP to support low-powered devices [15].

Although Contiki is an operating system for IoT devices rather than a central hub, it offers some interesting functionality. It shows that it is possible to have modular network support (in the form of
5.2. Network techniques for home automation

replaceable services) and shows that even the recent IETF standards for IoT devices can be supported in this way [16].

Mihini
The Mihini project is an open source framework which provides an application environment for IoT devices. It does this by exposing a high level Lua API running on top of Linux. The Mihini project was started by the Eclipse M2M working group with the goal to enable easy development of Machine-to-Machine applications. The framework manages data, I/O, devices, and applications and contains a REST API to control this management [17, 18]. Although Mihini aims to simplify IoT application development by providing an application container, applications themselves are responsible for managing application lifetime, configuration and communication protocols [19]. Mihini can be used on a Raspberry Pi, showing that a Lua environment on top of Linux is a feasible approach for the hub software.

Heimcontrol.js
Heimcontrol.js is an open source, plugin-based home automation solution built with Node.js. It is intended to run on a Raspberry Pi, but allows communication with an Arduino to which other devices can be attached. The feature that makes Heimcontrol.js stand out is its plugin architecture, requiring a plugin for every supported device [20]. At the moment of writing, Heimcontrol.js development seems to have come to a halt. Regardless, its plugin architecture, albeit incomplete, is a useful proof of concept and shows the feasibility of a Javascript-based plugin system.

Node-RED
Node-RED is an open source visual coding tool that is being developed by IBM with the IoT in mind. Like Heimcontrol.js, Node-RED is built on Node.js and can run on a Raspberry Pi. It allows users to create systems, called “flows”, by chaining blocks of javascript code, called “nodes”, to each other. Nodes and flows can be shared between users, enabling people to share code to control devices or entire systems with ease [21]. In light of this research, nodes can be seen as plugins, and a flow as a plugin which depends on the chained nodes. Node-RED is designed for users to easily combine existing blocks of functionality, but in doing so complicates development of nodes for controlling devices. However, its method of stacking or chaining plugins might pose useful for the software solution.

5.2. Network techniques for home automation

Many standardized solutions exist for device discovery, networking and data interchange. The large number of available standards, however, has caused fragmentation in the home automation market, with different devices often using different standards. Moreover, some vendors choose not to adhere to an existing standard and develop their own technology, further complicating a uniform communication interface. An overview of some standards commonly used in TCP/IP-enabled home automation devices is given.

5.2.1. Device discovery protocols
Device discovery, described in section 5.4, is the first step toward controlling devices on the local network. Unfortunately, there is not a single widely-adopted standard for device discovery. Instead, many different competing standards co-exist, with some devices supporting multiple standards, whereas others may not support any device discovery standard. Some device discovery techniques that have been applied to home automation devices are described below.

UPnP
Universal Plug and Play (UPnP) is a network protocol that allows zero-configuration networking and device discovery, to deliver seamless device connectivity. UPnP technology targets, among others, home networks. Using UPnP, devices can dynamically join a network, announce their name, convey their capabilities upon request, learn about the presence and capabilities of other devices, and leave a network smoothly and automatically without leaving any unwanted state information behind [22]. UPnP technology is built upon connection and communication standards, such as IP, TCP, UDP, HTTP, and XML. UPnP uses the Simple Service Discovery Protocol (SSDP) to find interesting devices [23].
Although UPnP is not directly applicable to low-power networks because of its complexity, it is used to create simple, complete, extendable and unified internet of things networks [24–26].

Third party discovery
Some vendors support discovery of devices in a local network through a cloud platform. Devices make use of these services by advertising their local IP address, possibly along with other relevant information, to the cloud platform, after which other devices in the same network can request the advertised information from the cloud. Philips, for example, offers a cloud discovery service called N-UPnP for its Hue product line [27].

SNMP
The Simple Network Management Protocol (SNMP) is a protocol that monitors and manages devices in IP networks, originally standardised by the Internet Engineering Task Force (IETF) in 1990 [28]. Although SNMP is the de facto network management method, application of SNMP environments in environments with resource limited devices is challenging [29]. Nevertheless, various extensions of SNMP have been used in home automation. [30].

mDNS and DNS-SD
Multicast DNS (mDNS) [31] and DNS Service Discovery (DNS-SD) [32] are two specifications that are part of the IETF’s zeroconf initiative, enabling communication between devices in the absence of fixed infrastructure. mDNS uses DNS queries to a special multicast address to resolve names among hosts on a local network. DNS-SD is a set of naming conventions which are used to represent services in DNS records. The combination of mDNS and DNS-SD can be used to discover devices on the local network and query which services hosts offer without the need of managed DNS services [33]. mDNS and DNS-SD are widely used in IP networks and are feasible solutions for smart objects as found in home automation networks [34].

IP range scanning
The final method to discover devices is the brute-force method IP range scanning. Some devices, such as the iKettle 1, use this method. This is not an efficient method, but can be applied in the absence of any other discovery protocol.

5.2.2. Application layer protocols
Although many applications on the internet have reached mutual agreement on which communication protocols to use, such as HTTP powering nearly all of the world wide web, no such agreement exists in home automation networks. Whereas some vendors opt to use well-known but potentially suboptimal application layer protocols such as HTTP, others develop their own solutions. Meanwhile, new protocols dedicated to IoT applications, such as CoAP, are being developed. Application layer protocols that are currently widely used in home automation devices or are believed to gain widespread support, including HTTP and CoAP, are described below.

HTTP
The widespread use of web technologies on the internet has recently sparked interest in the Web of Things (WoT), in which web technologies are applied to provide communication in the Internet of Things. The Hypertext Transfer Protocol (HTTP) [35], commonly used on top of TCP, is the well-known application layer protocol used on the world wide web. Despite the complexity of implementing a full HTTP/TCP/IP stack on devices with limited available resources, many home automation devices now include an embedded web server and a RESTful architecture, allowing for simple and uniform communication between devices [25, 36, 37].

CoAP
The Constrained Application Protocol (CoAP) [38] is a transfer protocol designed for devices and networks with few resources, such as low-power networks. CoAP provides the same basic set of services as HTTP, but is less complex and thus more suitable for devices with limited resources. Whereas HTTP operates over TCP, CoAP is typically implemented on top of UDP. CoAP also provides device discovery and an ‘observer’ pattern, allowing devices to push notifications to clients [39].

---

1 iKettle Wifi Kettle - http://smarter.am/
5.3. Used programming languages

Because of the large impact the choice of implementation language has on flexibility (4.1.1), performance (4.1.4) and simplicity (4.1.2), careful consideration should be made before choosing a language. Five potential implementation languages, all widely used in existing plugin systems, are reviewed in terms of performance, flexibility and simplicity.

C

The C programming language is a low level language and is currently used by HomeWizard for the development of their products. Its relatively low level of abstraction and easy access to hardware make it possible to write very efficient programs. The low level of abstraction comes at the cost of being a more difficult language. Since C is generally compiled directly to machine code, it can be run on Linux without the need for additional environments, making it a lightweight language.

Java

The Java programming language is regarded a good candidate for embedded systems, offering a higher level of simplicity than C because of dynamic class loading, garbage collection, and multithreading. On the other hand, as a result of these features, the Java Virtual Machine requires additional computational resources, making Java less suitable than C as an embedded programming language.

JavaScript (Node.js)

Node.js is a runtime environment providing non-blocking I/O and an event-driven model for the development of network applications. Node.js programs are written in javascript. Its non-blocking I/O enables Node.js to deal with a vast amount of simultaneous network connections. Node.js is fast enough to be used for home automation applications like Heimcontrol.js (described in section 5.1) and...
has a large repository of existing code, including libraries for interacting with some home automation devices. However, embedding javascript and binding javascript to native code are relatively difficult. Moreover, the asynchronous callback-based model employed by Node.js may be confusing for people with little programming experience. Furthermore, Node.js primarily aims to provide amounts of throughput far beyond what is required by HomeWizard [50].

Lua
Lua is a lightweight embeddable programming language with a relatively simple C API. Lua can be used as a very simple configuration language, because it does not require much punctuation. Lua has a small footprint and can easily be combined with other languages [51]. In comparison to C, Lua requires less proficiency in programming. Lua can be used as a plugin language because of its accessibility and extensibility. Simple plugin features could be written in Lua, while more complex features could be implemented in C. It is feasible to use Lua on a Raspberry Pi, as is shown by the Mihini project in section 5.1. Within HomeWizard, some developer experience with Lua is available.

XML and JSON
Since XML and JSON are easy to understand and human-readable languages, they could provide really simple plugins and simplify debugging. Minimal programming skills are required to develop a plugin using only XML or JSON. However, these languages do not provide any logic or arithmetic, which entails limited possibilities. Using XML and JSON, plugins are not extensible and therefore do not comply with the most important design goal ‘Flexibility’.

5.4. Device control steps
To be able to control a device, certain control steps are required from the central hub. Not all steps are required for all devices. An overview of control steps is given below.

Step 1: Device discovery
According to Edwards [33], discovery is “a mechanism for dynamically referencing a resource on the network”. Without knowing which devices are connected to the network, and how to reference these devices, the HomeWizard hub cannot communicate with any devices. The central hub must thus dynamically discover how to reference IP-enabled devices on the local network. IP-enabled devices connected directly to the local network can be discovered directly. Any gateways that are found during discovery will be asked to provide a list of devices connected to them. At the end of this step, the central hub knows where on the network the desired devices are located. This step can also be performed without the intent of controlling a specific device. In that case, the hub uses this step to get an up-to-date state of the network.

Step 2: Get device properties
The hub has to request the properties of a device when it does not (completely) recognize it. This device has been discovered during the discovery step, or has revealed its existence with a broadcast when it entered the network. Upon receiving the request for device properties, it will give a (protocol specific) response with the information needed by the hub.

Step 3: Setup connection
Some devices will require a connection with the central hub in order to function. This might be to establish a session or persistent connection between the hub and the device. This connection will be initialized on the initiative of either the hub or the device. The hub can use the plugin to determine what protocol has to be used for setting up this connection.

Step 4: Setup device registration
Initial registration may be required for some devices before they can be controlled. This is mostly due to security concerns. However, some devices might need to configure themselves to the network in order to function properly. Registration can involve the press of a physical button paired with a network message, the device creating its own temporal WiFi network to which the hub needs to be connected, or can be entirely automated.
5.4. Device control steps

Step 5: Communication with the device
The device is ready to communicate with the hub once it has been discovered, recognized, connected, and registered. IP enabled devices will communicate directly over the local network. Devices unable to use the Internet Protocol will communicate with an intermediate IP gateway, which will relay the communication to the central hub. The hub can send control instructions to the device; the device can send data to the hub. The encoding of the data is defined in the plugin, as are the types of requests and responses pertaining to the device. If communication is encrypted the encryption method will have to be defined in the plugin.

Step 6: Close connection
When the communication is over and the connection is no longer needed, the connection will be closed. The device will execute potentially received instructions or, if it is a simple sensor, it will go back to a sleep state to conserve energy. The hub can connect with the device when it needs to communicate with it again and vice versa.
The iKettle implementation

The iKettle implementation is the result of the first scrum cycle, based on the research of the previous chapters. The development of this implementation is guided by the design of plugins, to obtain the plugin development experience necessary for reaching the ideal plugin design. To be able to facilitate even the most unusual of devices, the iKettle implementation is focused around a fully functional plugin implementation for the iKettle, an exotic device ignoring all standards. Designing a plugin for such a highly unusual device reveals potential pitfalls and showstoppers in an early development stage, leading to a complete final product.

This chapter describes the bottom-up implementation of the iKettle during the first scrum cycle. The motivation for the design is given in section 6.1, followed by a detailed explanation of the plugin's structure. The architecture of the core components of the iKettle implementation is discussed in section 6.2. The iKettle implementation makes use of a dependency manager, which is detailed in section 6.3. The testing and documentation methods of the prototype are explained in section 6.4. Section 6.5 and section 6.6 describe problems encountered during the implementation of the first prototype and focus points for the next prototype, respectively.

6.1. The iKettle Plugin

Starting development by implementing a plugin for an exotic device gives a thorough understanding of the required functionality of a device plugin. The bottom-up methodology of writing a specific plugin before commencing development of the software system stimulates agile development and a fully functional implementation early in the development process. Therefore, the first prototype is built around a plugin for the iKettle, a WiFi-controlled device ignoring all standards.

The iKettle is a WiFi-controlled kettle with a proprietary TCP-based API. Since the HomeWizard hub can connect with the iKettle directly via TCP/IP, this device is best categorised as a single device as described in section 2.2. Although the device's manufacturer has no public API documentation, Cox [52] has documented the device's control instructions used by the smartphone application supplied by the manufacturer. The iKettle app uses an IP range scan on port 2000 to discover devices on the local network, after which custom TCP packets are used to communicate with the device. The iKettle sends asynchronous status messages over all open connections, e.g. to notify clients when the water in the kettle has reached the desired temperature. Connected applications must thus continuously listen to incoming messages from the iKettle.

The iKettle plugin specifies actions to set the target temperature, control the device's 'keep warm' function or turn the device on or off. Moreover, the plugin specifies triggers indicating that the device is on or off, that the kettle has reached its target temperature, that there is a problem with the kettle, that the kettle has been removed, or that there are changes in the device's keep warm function. The complete contents of the iKettle plugin are included in appendix A.

6.1.1. Implementation language

As the language used for writing plugins has a large influence on flexibility, simplicity and performance, careful consideration of potential languages is required. Table 6.1 contains a comparison of the potential
implementation languages described in section 5.3 in terms of flexibility (4.1.1), simplicity (4.1.2), performance (4.1.4), maintainability (4.1.5), and the available ecosystem.

C, being a system language, provides far more flexibility and performance than Java, but comes at the cost of increased complexity due to memory management. Although C does not provide object-oriented language constructs like Java, hardware developers generally have more experience with C. Large amounts of existing code are available for both Java and C, but few open source home automation projects are written in either language.

JavaScript and Lua, both dynamically typed languages, offer more flexibility than Java in how plugins are written. The ease with which C code can be used in Lua makes Lua even more flexible than JavaScript, while the high level of both languages offers more simplicity than Java and C. Although JavaScript is more widely known than Lua, Lua is easier to learn with its small size and simple type system. Moreover, HomeWizard developers have experience with Lua. The main disadvantage of JavaScript is its large performance footprint in terms of both speed and memory usage, whereas Lua is almost as fast as Java while having a smaller memory footprint. Many existing libraries are available for JavaScript, including some open source home automation projects, whereas Lua has a considerably smaller ecosystem. Both languages have prior art in home automation projects. XML and JSON, although ideal for writing plugins from a simplicity perspective, do not offer enough flexibility to write complex plugins.

Since flexibility and simplicity are more important design goals than performance and maintainability, Lua is more suited for plugin development than the other listed languages. With Lua, simple plugins can be written much like configuration files, whereas complex behaviour can be implemented using functions. Therefore, Lua is the language of choice for implementing plugins. Lua is also used as the implementation language of the software system itself for two reasons. Firstly, the use of a single language for all components increases maintainability as well as performance, as no switches between languages have to be made. Secondly, the simplicity and flexibility of Lua allow for rapid development while barely affecting performance. To ensure a low performance footprint, the LuaJIT just-in-time compiler for Lua is used to run Lua scripts. LuaJIT boasts high performance and an unmatched low memory footprint, while being fully compatible with Lua 5.1 [53]. Moreover, LuaJIT is available for Linux running on ARM processors, meeting requirements 5 and 6.

### 6.1.2. Abstractions in plugins

To simplify plugin development (4.1.2: Simplicity), plugins should only be required to specify operations that are directly related to controlling a device. Therefore, required functionality not directly related to controlling a device, should already be implemented into the system. This is functionality such as implementations of commonly used communication protocols. Control operations should, where feasible, be described using simple configuration statements. For example, after declaring which communication protocol should be used, control instructions can be sent using a single statement. Moreover, plugins should not have to distinguish between devices; it should be possible to control any number of devices with a plugin that is written as if it controls a single device.

### 6.1.3. Plugin separation

To prevent plugins overwriting data of other plugins, or of the core system, each plugin is given its own environment (4.1.3: Security and requirement 9). The environment of a plugin is a table, which is
accessed by a plugin as if it is the global scope. Functions and variables declared in a plugin are placed inside the environment of that plugin, thus preventing overwriting data of other plugins. Likewise, plugins cannot easily read values defined by other plugins, or by the core system. Each module that is included into a plugin using the require function that is customary in Lua is placed inside the environment of that plugin, allowing the module to access data specific to that plugin, such as which device it is currently operating on.

In order to run multiple plugins for multiple devices at once while ensuring responsiveness (4.1.4: Performance), some form of multitasking is required. Lua has native support for lightweight coroutines, allowing nonpreemptive multitasking with little overhead. Plugins are run concurrently by yielding to a controller on each blocking network operation, only to be resumed when the network operation completes or a timeout occurs. Although this method of multitasking has very little overhead and does not require additional measures to ensure data integrity, a plugin can cause starvation in all other plugins as well as the core by executing long-lasting operations.

6.1.4. Plugin development simplified
Implementation details of commonly used functionality, such as TCP connectivity or HTTP communication, are hidden from the plugin developer through the use of modules. The iKettle plugin uses the modules tcp, for TCP connectivity, and ipscan, for device discovery. ipscan, in turn, relies on the arp module to determine a device's MAC address. If no module is available for certain functionality, plugin developers can create a new module to provide that functionality, after which other developers can use that new module.

To offer developers an environment to implement control steps requiring even less engineering effort, some libraries implementing a standard interface can be included using simple configuration statements. For example, a plugin can specify that devices should be discovered using an IP range scan on TCP port 2000 using the statements shown in listing 6.1. The system interprets these lines, includes the module for the specified protocol, and calls the discovery function specified in that module with 2000 as the port. If the discovery method of a device is not available in a module, plugin developers can instead define a discovery function themselves.

```lua
discovery = {
    protocol = 'ipscan',
    port = '2000'
}
```

Listing 6.1: The simplest implementation of the discovery control step. By defining information required for the discovery process the system can automatically execute the discovery process.

Communication with devices can be achieved in the same manner. Modules implementing a send and a receive function can be used by specifying the name of the module as communication protocol, after which a simple call to send or receive can be used from within the plugin. All actual communication logic is handled inside the module; the plugin developer only has to specify the communication data (i.e. control instructions or device responses).

Because the iKettle API is largely illogical and inconsistent, some processing is required to translate commands, such as to change the target temperature, to control instructions. The necessary processing is performed in functions, using simple if-then-else statements. An example of such a function, used to set the temperature of the kettle, is shown in listing 6.2. Although a small amount of programming is required to specify such functions, simple if-then-else structures are sufficient for most communication functions.

Receiving the asynchronous status messages sent by the iKettle requires some more programming. The status messages are sent in the format sys status <code>, where <code> is variable (of the form '0x' followed by one to four digits). Because some protocols, such as TCP, do not specify message boundaries, plugin developers must be able to specify the structure of messages, while not being forced to explicitly list every possible message. Therefore, developers can specify patterns and register callback functions to be invoked whenever the received data matches a pattern. The code listed in listing 6.3 listens for asynchronous status messages sent by the iKettle by registering a callback function for the pattern 'sys status (0x%d+)\r', where (0x%d+) is used to extract strings starting with 0x, followed by one or more digits. The callback function receives two arguments, the first being
6.1. The iKettle Plugin

### Function Implementation

```lua
function setTemperature(temp)
    if temp < 80 then
        send("set sys output 0x200\n") -- set temperature to 65 degrees
    elseif temp < 90 then
        send("set sys output 0x400\n") -- set temperature to 80 degrees
    elseif temp < 100 then
        send("set sys output 0x2\n") -- set temperature to 90 degrees
    else
        send("set sys output 0x80\n") -- set temperature to 100 degrees
    end
end
```

Listing 6.2: A function can define how actions should be processed. Logic can be implemented using simple IF-THEN-ELSE structures.

### On-Receive Function

```lua
onReceive('sys status (0x%d)\n', function(response, r)
    if r == '0x100' then status.temperature = 100
    elseif r == '0x90' then status.temperature = 90
    elseif r == '0x80' then status.temperature = 80
    elseif r == '0x65' then status.temperature = 65
    elseif r == '0x11' then status.warm = true
    elseif r == '0x10' then status.warm = false
    elseif r == '0x5' then status.on = true
    elseif r == '0x0' then status.on = false
    end
end)
```

Listing 6.3: Responding to asynchronous status messages. Updates to variables prefixed with status are used to update triggers.

As described in chapter 2, capabilities of devices must be exposed to the HomeWizard hub in terms of actions and triggers. Exposing these capabilities from within a plugin is done specifying the name and the description of an action or trigger, as in listing 6.4. The name of an action corresponds to a function with the same name in the plugin that implements the action, e.g. the "setTemperature" action corresponds to the setTemperature function described in listing 6.2. The name of a trigger corresponds to a field with the same name in the status table; any update that a plugin makes to a variable prefixed with status is registered as an update to the trigger. The "on" trigger, for instance, is updated to true when the statement status.on = true is executed, as is done in listing 6.3.

```lua
action "setOn" "Turn the device on or off."
action "setTemperature" "Set the temperature to %d degrees celsius."

trigger "on" "device is %b"
trigger "temperature" "temperature is set to %d degrees"
```

Listing 6.4: Defining actions and triggers in a plugin.

### Implementing Complex Functionality

Plugin developers in need of more complex control than is provided by available modules can write functions implementing the required behaviour themselves, as well as write new modules to use in their plugin. Instead of declaring which discovery or communication modules should be used, plugin developers can write discovery or communication functions specific to their device. Moreover, the flexibility of Lua allows complex operations on data, if needed. The iKettle plugin, for instance, uses bitwise operations to extract status information from an initial status message, in which the device
status is encoded using bit fields. These operations are shown in listing 6.5. The only requirements for plugins are that they must define a discovery method, be it a configuration table or a custom function, and define actions and triggers. Moreover, the device status should be stored in variables prefixed with status. Apart from these requirements, plugin developers are entirely free in how they write their plugin.

onReceive('sys status key=(..)\r', function(response, key)
    local num = key:byte()
    status.on = bit.band(num, 1) == 1
    status.warm = bit.band(num, 2) == 2
    status.temperature = bit.band(num, 32) == 32 and 100
    or bit.band(num, 16) == 16 and 90
    or bit.band(num, 8) == 8 and 80
    or bit.band(num, 4) == 4 and 65
    or 0
    return true
end)

Listing 6.5: Bitwise operations to extract status information from iKettle status responses.

6.2. Core architecture

The core of the software solution exists of several components which together facilitate the utilisation of the plugins. The most important components are the Executor, the DeviceList and its Devices, the Plugin class, and the Context Injector. The Executor component handles external requests, such as the existing HomeWizard software instructing the system to add a new ip device. The Plugin class delivers the main functionality required for plugins. This Plugin class is used by the Context Injector when loading manufacture-written-plugins. The operation of the Context Injector and Device List are a little more complex.

Figure 6.1: A class diagram of the core of the iKettle implementation
Executor
The executor component is able to receive external requests. The HomeWizard software can send commands to the executor to operate the system. Main functionality is installing new devices and exposing triggers/actions to the HomeWizard software. The executor uses the Context Injector and the Plugin class for its operation.

DeviceList and Devices
The Device List keeps track of the connected devices. It handles the registration of new devices. Devices are kept in an array and can be found using unique numeric identifiers. Device objects have a ‘parent’ and a ‘children’ field in which it keeps track of its parent and children. The hierarchy of the Device objects is derived from their place in the network: a device that is connected through a gateway will have this gateway as its parent. Each device has a hardware identifier. This identifier, in combination with the chain of devices between the HomeWizard hub and the device act as a unique identifier. Devices can be deleted from the Device List through a call to the Executor.

Plugin class
The plugin class is the focal point of the system. Almost every other module either extends this class’ functionality or is used by this class. The plugin class provides a layer between modules and the core of the system. The plugin class gives modules access to predefined functions (i.e. action and trigger) and provides an easy connection to the plugin’s environment. Furthermore, since plugin developers can define protocols in multiple ways, the plugin class interprets how these protocols are defined.

Context Injector
The Context Injector takes care of loading plugins. To prevent a plugin polluting the global space, the environment or context of the plugin is set to a unique context for the loaded plugin. Any globals declared in the plugin will be placed inside that context. Moreover, the Context Injector configures the context of a plugin in such a way that, from the plugin’s point of view, the global variable ‘device’ always points to the device it is handling at the moment. This allows easy operations on devices, without having to pass devices around between functions within the plugin. To protect important device-specific information, a plugin can store data in the status table of the device.

6.3. Dependency Management
To achieve the desired flexibility (4.1.1), modules may depend on other components providing certain functionality. The tcp module, for instance, depends on LuaSocket, a component providing network support for Lua. The iKettle implementation uses the LuaRocks package manager [54] to install most of its own dependencies at runtime.

LuaRocks can install packages at runtime, which allows future HomeWizard developers to easily implement an installer for plugins. HomeWizard could set up its own LuaRocks repository with plugins and modules, alongside the default LuaRocks repository with existing modules such as LuaSocket. Moreover, LuaRocks automatically searches and installs dependencies of packages. This way, the end user would only have to select the plugin to install, while the package manager installs the plugin and all of its dependencies.

6.4. Documentation and Testing
The source code of the iKettle implementation is documented using documentation comments extracted with LDoc [55], the most widely used documentation generator for Lua. All modules and public functions in the core, the ipscan module, and the tcp module are documented, adhering to requirement 10. The generated documentation is included in the git repository in HTML format.

All core components are unit tested using the busted [56] unit testing framework. The iKettle implementation contains 103 tests, with a combined test coverage of the core components of 90.10%, sufficient to adhere to requirement 11.

6.5. Encountered problems
During the work on the initial prototype, the issue of separation emerged. As plugins will be written by third parties, their level of quality is not assured. To prevent the system from crashing completely on
faulty plugins, some level of separation is required. Ideally, each plugin would have its own process, granting the best separation. However, some functionality would have to be duplicated in each process, increasing memory usage of the system. An alternative to using a new process for each plugin is the use of multiple Lua states. A Lua state contains all data used by Lua. Using multiple Lua states thus prevents any use of data from other plugins. Unfortunately, the overhead of Lua states is still significant, although less than that of a subprocess. A third alternative is separating the system into two processes. The core system will have its own process and all the plugins will run together in the other process. This eliminates the problem of overhead, but one faulty plugin can still crash all other plugins. Nonetheless, although the core is still running, without active plugins it has no real functionality. Separation is not been realised in this prototype, but is postponed to the next prototype.

6.6. Next sprint
The next prototype’s biggest feature will be communication with devices behind gateways. This is HomeWizard’s primary interest. Now that the core mechanisms of the system are done, this feature has the highest priority. Modules that support some of the common protocols as discussed in section 5.2 should be developed to give the prototype some ‘off-the-shelf’ functionality. Furthermore, a decision has to be made on the separation issue.
The Pixar Prototype

Although the iKettle implementation provides functionality for controlling single TCP/IP devices, gateways used to reach multiple devices cannot be controlled. Therefore, the prototype developed during the second two-week scrum cycle, called 'the Pixar Prototype', focuses on communicating with devices via gateways. The implementation of the second prototype is driven by a plugin for a simple RF-controlled switch, turning on and off a lamp much like Pixar’s mascot Luxo, Jr. (hence the name). The switch, in turn, is controlled via a HomeWizard hub acting as gateway. The setup of devices is depicted in figure 7.1.

Figure 7.1: Setup of devices in the Pixar Prototype. The Pixar Prototype software itself, although intended to run on a HomeWizard hub, runs on a laptop computer. From the Pixar Prototype, instructions are sent to the HomeWizard hub, acting as gateway to a 433 MHz RF switch. The switch, in turn, controls the supply of power to a light bulb.

One of the most important features of the Pixar prototype is the kernel-level separation of plugins, which is explained in section 7.1. The new HomeWizard plugin is detailed in section 7.2, followed by important changes in core components in section 7.3. Section 7.4 specifies the progress made concerning documentation and testing. Some problems encountered during the implementation of this prototype are discussed in section 7.5. During this sprint, the team received feedback from the Software Improvement Group, on which is reflected in section 7.6. Plans for the next sprint are presented in section 7.7.
7.1. Separation of processes

As described in section 6.1.3, the first prototype executed all plugins as well as the core in a single process. Barriers between plugins and the core, as well as between plugins, were imposed by executing each plugin inside its own environment. Although separation by means of different environments provides superficial separation in terms of data access, proficient programmers can easily circumvent the sandbox imposed on a plugin through Lua's debug library or its C API, while an endless loop, be it accidental or malicious, is enough to make the entire system unresponsive.

To overcome the extensive shortcomings of a single-process plugin system, the second prototype runs each plugin in its own process, taking advantage of kernel-level separation and scheduling. Such process-based separation improves security (4.1.3), responsiveness, and stability (4.1.4). Although separation between processes is not water-tight on most systems, it does provide reasonable barriers to limit interference between plugins as prescribed by requirement 9.

Despite the advantages of the multi-process architecture of the Pixar Prototype in terms of security and stability, splitting up the system into multiple processes does have performance drawbacks. Despite the small memory footprint of LuaJIT, first attempts to run plugins in separate processes resulted in a unique set size (USS) of nearly 1 MB per new process, which is a considerable amount on the resource-constrained hardware used in the HomeWizard hub described in section 2.1. Reducing the number of dependencies included in plugin processes by forking each plugin process from a small-size host process rather than directly from the main process, together with reducing the size of dependencies included inside plugin processes, reduced the unique set size of fully functional plugins to 300 to 400 kB. This overhead was found to be acceptable by HomeWizard developers.

7.1.1. Three types of processes

The system is split in three types of processes: the Master process, Slave processes (also called Plugin processes), and the PluginHost process. The Master process entails almost all functionality of the system. It is the central core of the system and processes all communication streams, keeps track of devices with the DeviceList and controls plugin processes with the PluginController. Each plugin runs in its own process, with all plugin processes communicating with the master process. In order to give processes minimal memory overhead (4.1.4: Performance), forking is used to create new processes. When loading plugins, a new process is forked from the PluginHost process. The PluginHost's process is forked from the main process as early as possible in order to contain only essential dependencies. This way, when a new plugin is loaded, the newly created process has a small memory footprint. This forking scheme is depicted in figure 7.2.

Inter-Process Communication

Slave processes communicate with the master process via a shared Inter-Process Communication (IPC) module. Slave processes connect to the master process immediately after their creation, setting up a two-way communication channel between the slave and master processes. The IPC module acts both as server accepting incoming Remote Procedure Calls and as client executing Remote Procedure Calls. Since the master and the slave process use the same IPC module, the slave process can invoke functions on the master and vice versa. To keep control over communication streams and improve security, slave processes cannot communicate with each other and are unaware of each others existence. The methods in figure 7.4 which are accessible via IPC are annotated with «IPC».

A custom IPC module using MessagePack-encoded messages over TCP sockets is used in the software system because no stable standalone IPC library for Lua is available. Since the IPC interface is only used internally, easily available, well-documented and regularly updated components are used rather than adhering to commonly used RPC standards, allowing for a relatively small IPC module of 132 code lines (4.1.5: Maintainability).

Process overview

To give an overview of the processes and their intercommunication, a few actions are depicted in the flow diagram in figure 7.3. The vertical lines are the process lifelines. The diagram shows how processes are created and how they collaborate to perform actions.

As can be seen in the diagram, the PluginHost is forked early in the process and is later used to create a new plugin process. Intercommunication between processes is done using the IPC class. When
7.2. The HomeWizard gateway plugin

To control the switch and thus the lamp via a gateway, the system must know how to control that gateway. Although the HomeWizard hub supports a wide range of devices, all of which can be controlled via HomeWizard’s proprietary HTTP-based API, the HomeWizard hub cannot be classified as a generic gateway. Instead, the HomeWizard hub is a dedicated gateway for a large variety of devices. Each supported device is assigned a category with a specific command set. Switches, for instance, are controlled by sending an HTTP GET request to /sw/<switch_id>/on or /sw/<switch_id>/off, where <switch_id> is the identifier of the switch used by the HomeWizard hub.

The choice to use the HomeWizard hub as an intermediate gateway is based not only on its availability, but also on its functionality. The HomeWizard hub, supporting many different command sets for an ever-increasing range of devices, has an inconsistent API that is being expanded with nearly every new device that is supported. Although it is possible to implement all current HomeWizard functionality in a single plugin, maintaining this plugin would be cumbersome due to its large size and the continuously expanding functionality of the HomeWizard hub. Therefore, a plugin describing only the core functionality of the HomeWizard hub (i.e. communicating with the device itself) is implemented, along with plugins providing functionality for devices behind the HomeWizard hub.

The Pixar Prototype contains the implementations of a plugin for the HomeWizard hub and of a plugin for a switch operating on the 433 MHz RF band, operated via a HomeWizard hub. The HomeWizard plugin describes discovery and communication with HomeWizard hubs. Moreover, it implements functions that can be used to discover and communicate with ‘children’, or devices behind the gateway. The HomeWizard.switch plugin defines which commands should be sent to the HomeWizard hub to turn a switch on or off. Both plugins are included in appendix B.
7.2. The HomeWizard gateway plugin

7.2.1. Gateways and children
Implementing support for devices acting as gateways to other networks has far-reaching consequences for both the core and plugins. As a gateway plugin may control multiple physical gateway devices, which in turn may represent multiple devices on another network (henceforth called children), device management as used in the iKettle implementation does not suffice. Moreover, gateways may support many different devices and may thus have a large number of possible control instructions.

The simplest approach to supporting gateways and their children is to describe both the functionality of the gateway itself and that of its children in a single plugin. However, plugins for gateways supporting many devices and control instructions will become unwieldy and hard to maintain. Moreover, if one were to chain gateways behind one another, the plugin for the first gateway would have to include not only the instructions for itself and all possible devices in its network, but also the instructions for all devices in the network behind the second gateway. In extremis, this would require an infinitely large plugin.

To allow more flexibility in creating networks of devices, a plugin for a generic gateway can be split into a plugin for the gateway device itself, and plugins for one or more devices behind that gateway. This allows users to install plugins that are required, without having to pull in instructions for the entire universe of possible devices behind a gateway. For instance, one can install only the functionality for a switch behind a HomeWizard hub, without having to install functionality for weather stations, cameras, vacuum cleaners, etc.

Splitting plugins, however, requires functionality in the core system to allow plugins to cooperate, e.g. the HomeWizard plugin telling the switch plugin which switches are connected to the HomeWizard hub, or the switch plugin telling the HomeWizard plugin which data to send to the HomeWizard hub. One possible implementation of cooperation is to load actions and triggers defined in plugins for children directly into the plugin of the parent at runtime, essentially as a mixin. However, this requires careful device management, as the system must keep track of which actions and triggers are valid for which
7.2. The HomeWizard gateway plugin

discovery = {
    protocol = 'gateway',
    gateways = 'HomeWizard', -- May list multiple gateways, e.g. { 'HomeWizard', 'HomeKit' }
    category = 'switches',
    type = 'switch',
    prefix = 'sw'
}

communication = {
    protocol = 'gateway'
}

Listing 7.1: Discovery and protocol specification of a device behind a gateway.

devices. Moreover, plugins must keep track of both gateway devices and their children, which becomes especially hard when gateways get chained. Therefore, a different approach to combining plugins for gateways and their children is taken in the Pixar Prototype. Rather than loading all functionality into a single plugin, each plugin is executed separately. Discovery and communication operations performed in the plugin for a child device are forwarded to the plugin for its parent, which describes how the operations should be sent to the gateway device by implementing a function called discoverChildren, and optionally functions called sendTo and receiveFrom. The core keeps track of all child devices created during discovery and passes those devices to the sendTo and receiveFrom functions when necessary. Because each plugin operates on at most two devices (one gateway device and one child device), separate plugins offer simpler device management than plugins that are mixed together.

7.2.2. Implementing gateway communication

Discovery of devices behind a gateway is always initiated from the plugin for the child device by using the gateway discovery protocol, not from the gateway plugin itself. The main reasons for this are that gateway plugins may not know how to discover each child device type, and that otherwise the gateway plugin may not know which plugin must be loaded for which type of device. The end user must thus explicitly install a plugin for each device type behind a gateway that they want to use and cannot discover every device behind a gateway without installing the plugins for all those devices.

Listing 7.1 contains the discovery and protocol specifications of a switch behind a HomeWizard gateway. The specified gateway discovery protocol is built into the core, invoking the discoverChildren function on all devices of the type listed in the gateways field of the child plugin. The remaining fields in the discovery table, i.e. category, type and prefix, are passed to the discoverChildren function.

The discoverChildren function, which must be defined in a plugin for it to be used for gateway discovery, should perform device-specific discovery steps, optionally using additional information supplied by the child device. Each new device that is added by calling createChild is passed back to the child plugin. Listing 7.2 contains the implementation of discoverChildren of the HomeWizard gateway plugin. (Note that the ‘switches’ category contains not only switches; the ‘switch’ type is specified to ascertain that no dimmers are added.)

Communication with a device via a gateway is performed in a similar manner. Each call to send or receive made in a plugin for a child device is forwarded to the plugin for its parent device. If the plugin for the parent device includes a sendTo function, that function is invoked with the child device and the data to send as arguments. Otherwise, the data to sent is passed directly to the send function in the parent plugin. The HomeWizard sendTo function, listed in listing 7.2, assembles a message based on a device's prefix as specified in the discovery table, the identifier of a device used by the HomeWizard gateway, and the command sent by the child plugin. The call send 'on' on a device with HomeWizard identifier 3 in the HomeWizard switch plugin is thus sent to the HomeWizard gateway as an HTTP GET request to sw/3/on, which is exactly the command to turn a switch on.
function discoverChildren(args)
    local res = send 'get-sensors'
    for _, device in pairs(res.response[1.args.category]) do
        if not args.type or device.type == args.type then
            local child = createChild(device.name .. '_' .. device.type .. '_' .. device.id)
                child.id = device.id
                child.prefix = args.prefix
            end
        end
    end
end

function sendTo(child, command)
    return send(child.prefix .. '/' .. child.id .. '/' .. command)
end

7.3. Important changes and reuse of code

As a result of the separation of processes as described above, some changes have occurred in the core architecture. In order to facilitate the separation of processes, many classes have been split up, were combined, or have been altered otherwise. The core system has also been split into three 'packages' to provide clearer separation between components used only in the master process, components used only in slave processes and components shared by the master and slave processes. The most noteworthy changes, in comparison with the iKettle prototype, are described in this section. Class diagrams of both the Master and Slave have been included in figure 7.4 and figure 7.5 respectively.

Plugins & PluginController
The Plugin class has changed drastically. In the iKettle implementation, the Plugin class held all information regarding a plugin's capabilities. In this prototype, the Plugin class still exposes list of Actions and Triggers, but the actual Actions and Triggers are not kept in the Plugin class, but in the Plugin process instead. When the server receives a request to execute a certain action, this request is send to the relevant Plugin process using the IPC module and is handled there. The PluginController keeps track of running Plugin processes and asks the PluginHost process (using pipes) to fork a new process when a new Plugin process is needed.

Devices & DeviceThread
The Device class from the iKettle implementation has been split over the Master and Plugin processes: the functionality needed in the Master process is now in the Master.Device class, while the Plugin process uses the Slave.Device class to refer to Devices. Both classes inherit basic Device functionality from the shared.Device class. Since it is not desirable to pass all device data between processes, device data, such as the device.status field, is kept in the Plugin process. The Master keeps track of important device meta-data such as parent and children. A unique numeric identifier (the index field of the Device class) is used to refer to devices in communication between processes. Just as in the iKettle prototype, the master keeps track of all devices using the DeviceList. The Plugin process has a new class, the DeviceThread, which keeps track of all Devices under the Plugin's control. These Devices are all assigned to their own thread, enabling plugins to control multiple devices simultaneously.

Executor & PluginLoader
The Executor and ContextInjector from the iKettle prototype are now part of the plugin process. The Executor handles incoming IPC-calls from the master, to perform actions specified in the plugin. Since the separation of contexts between different plugins, as done in the iKettle implementation, became less important with the introduction of separation of processes, the ContextInjector is altered into the PluginLoader. The PluginLoader still takes care of loading plugins and setting up the sandbox to prevent pollution of the global space. However, every process now has only one sandbox and therefore the PluginLoader is simplified.
7.3. Important changes and reuse of code

Figure 7.4: Class diagram of the Master process. Methods accessible from other processes are annotated with «IPC». This diagram also represents the final prototype.

Figure 7.5: Class diagram of the Slave process. Methods accessible from other processes are annotated with «IPC». This diagram also represents the final prototype.
7.4. Documentation and Testing
In the second sprint, the documentation and tests created for the iKettle implementation were improved for the Pixar Prototype. The source code documentation of the Pixar Prototype covers all modules and public functions and is structured to reflect the different processes. The 188 unit tests cover 97.19% of the code of the system core.

7.5. Encountered problems
During the implementation of the Pixar Prototype, gateway plugins had no access to information specific to child devices. Device plugins can use a gateway plugin in order to control a device behind a gateway. Gateway plugins possibly need device-specific information about the child device it communicates with (e.g. a password or hardware identifier). However, the gateway and device plugins both run in their own process. Therefore, a problem arose when a gateway plugin wanted to use data from the device plugin.

One possible solution to this problem is sending the required device data to the gateway process with every communication call. This, however, would complicate plugin development for devices (behind a gateway), as the transmission of the data to the gateway would have to be implemented in the plugin. Moreover, sending the data on every call is a very inefficient. An alternative solution involved storing a copy of a gateway’s child devices in the gateway's process. This way, the gateway plugin can access required device information without demanding it from other plugins. Keeping a copy of all child devices came with a small memory cost, but was seen favorable over complicating plugin development. As a result, this solution was chosen and implemented in the system.

Another problem introduced during the implementation of the Pixar prototype, is the absence of a generic gateway. As mentioned in section 2.2, besides single devices and dedicated gateways, generic gateways may be present to control home automation devices. Unfortunately, we could not get hands on a generic gateway. Moreover, the client prioritizes dedicated gateways over generic gateways. Therefore, generic gateways are not yet supported in the Pixar prototype. Future work may consist of providing support for this type of gateways.

7.6. SIG feedback
The first SIG feedback points out that a few functions have a high complexity. SIG points out that it would like to see a maximum cyclomatic complexity of 5. However, the only actual function they mention, is an example function in iKettle.lua. This function is an example of how an unskilled programmer could implement a plugin. Therefore, it intentionally has a high complexity. The complete SIG feedback is included in appendix D.

For the next SIG submission, this kind of example codes and third-party libraries were omitted in order to receive feedback on the actual code. The code is checked on various metrics before sending it to SIG for the second round of feedback. Only a tiny percentage of the code has a cyclomatic complexity of 5 and more than 85% has a complexity lower than or equal to 3. Other applied metrics include test coverage, lines of code, number of arguments and coupling.

7.7. Next sprint
The next sprint is the final stage in delivering the final prototype to the client. As the most important functionality is already included in the prototype, the focus will be on doing some last improvements on the Pixar Prototype. During this last sprint, the prototype will be tested on the actual dedicated hardware. An acceptance test for plugin developers will be carried out and results will be determine the final changes to the prototype. Furthermore, a simple web-based front-end will be developed for demo purposes.
Outsourcery is the final prototype developed during the third and last sprint. Since the Pixar Prototype already contains the required functionality, Outsourcery is an improved version of the Pixar Prototype. As the product is released to the HomeWizard developers after this last two-week scrum cycle, Outsourcery focuses on delivering the ‘complete package’, including a simple demo interface and a dedicated developer tutorial. An overview of these and other features is given in section 8.1.

An important aspect of the last sprint is the user acceptance test, conducted with one of the developers of the HomeWizard hub. This test has two goals: it evaluates the functioning of the system, and it allows the HomeWizard developer to get acquainted with the system, facilitating easier integration of the software system into the HomeWizard hub. The user acceptance test is described in more detail in section 8.2. A reevaluation of the Software Improvement Group is discussed in section section 8.3.

8.1. Outsourcery
Since Outsourcery is the final prototype, the rough edges of the Pixar Prototype are polished to make Outsourcery fully fledged for delivery. Outsourcery includes a plugin for the Philips Hue, a simple web interface for demonstration purposes and various smaller improvements.

8.1.1. Philips Hue Plugin
The result of the final sprint includes a plugin for the Philips Hue, a highly specialized dedicated gateway. The Philips Hue lighting system consists of ZigBee-controlled lights that are controlled from a central bridge, accessible via a RESTful HTTP interface. The Philips Hue bridge is thus a dedicated gateway to a ZigBee network. Unlike the HomeWizard hub, the Philips Hue bridge supports only a few devices, all of which are lights. Implementing a plugin for the Philips Hue may uncover pitfalls specific to dedicated gateways that were not apparent for the rather generic HomeWizard hub.

Philips provides a server-mediated discovery service named N-UPnP, as described in Third party discovery (section 5.2.1). An HTTPS request to the discovery service results in a JSON-encoded response containing the bridges present on the local network. The Philips Hue bridge itself exposes a RESTful HTTP interface with JSON-encoded requests and responses, allowing re-usage of the HTTP and JSON modules developed for the Pixar Prototype. The developed Philips Hue plugin provides only the basic functionality to switch lights on and off and loop through colors (disco! mode). Additional functionality can easily be added in the future.

Architectural limitations
The Philips Hue bridge is, like the HomeWizard hub, a dedicated gateway. However, it differs from the HomeWizard hub in terms of the number of supported devices. While the HomeWizard hub supports a plethora of different devices, the Philips Hue only supports various kinds of lights. Although the different lights supported by the Philips Hue may support different actions, e.g. only some lights being able to change colors, there is much overlap in functionality between devices.

Creating a unique plugin for each type of light would result in large amounts of duplicate code for actions that are supported by multiple, if not all, devices. Turning a device on and off, for instance,
would have to be implemented in each plugin. Creating a single plugin for all types of lights, however, may expose more actions than a device supports, e.g. allow changing the color of a light that only supports white light. This can potentially cause confusing or problematic behaviour.

The current plugin system does not intuitively allow sharing functionality between plugins (although shared functionality could be implemented in a module), nor does it allow exposing actions and triggers on a per-device basis. Future work could improve sharing functionality through more elegant mixins, or per-device actions and triggers.

### 8.1.2. Web Interface

To demonstrate the potential of Outsourcery when integrated into the HomeWizard hub, the final prototype includes a simple web interface. The web interface demonstrates how devices and their actions and triggers can be used from a graphical user interface, allowing easy device management for the end user. However, the provided web interface is limited to controlling the current set of devices and is not generic enough for actual use on the HomeWizard hub.

To make Outsourcery usable for customers of HomeWizard, a more flexible and complete user interface should be created. It is advisable to give plugin developers more control over how parameters of actions are to be supplied by the end user in a graphical interface, e.g. specify that a continuous slider should be used for a dimmer, or that a slider with discrete points should be used to select the temperature of a kettle.

### 8.1.3. Polishing the product

Apart from a plugin for the Philips Hue and a web interface, the final prototype includes many small improvements and bug fixes to improve the usability of the prototype. Such improvements include an extended API for external components, i.e. other software on the HomeWizard hub, and the ability to stop running plugins. Moreover, Outsourcery was tested on the target hardware, and a tutorial for developing a plugin was created.

**Running Outsourcery on the HomeWizard hub**

The final sprint included testing Outsourcery on a Raspberry Pi to verify that the prototype adheres to the hardware limitations described in section 2.1. The test consisted of running the software on the Raspberry Pi running the Linux distribution Raspbian, and controlling every device currently supported: the iKettle, the HomeWizard hub, and the Philips Hue. Outsourcery was able to control these devices from a Raspberry Pi without noticeable delay and thus meets requirements 5 and 6.

**Tutorial: Developing a simple plugin**

A tutorial on plugin development is provided to assist the plugin developer in the user acceptance test. The goal of the tutorial is twofold. Firstly, the document is an indispensable part of the user acceptance test, providing the plugin developer with the basic required knowledge to develop a plugin. Secondly, HomeWizard can use the tutorial to aid manufacturers in developing plugins for their devices. This contributes to the delivery of the 'complete package'.

The tutorial goes through the development process one step at a time, using code examples to illustrate the process. It provides plugin developers a starting point and contains all knowledge required to successfully create a plugin. The tutorial is included in appendix C.
8.2. User acceptance test

The user acceptance test is the conclusion of both this sprint, and the project as a whole. The test, conducted with one of HomeWizard’s developers, provides important information for the completion of the final prototype. The test consisted of the HomeWizard developer implementing a plugin for the iKettle due to the lack of other available TCP/IP-enabled single devices. Using an available gateway device for the user acceptance test was found to be unsuitable for measuring the usability of the system for developing simple plugins, as gateways inherently require more programming than single devices.

The main goal of the acceptance test is to evaluate the functioning of the system. During the experiment, the developer is closely monitored and inquired for feedback. Key points of interest are:

- Time required for development;
- Intuitive use of plugin environment and pre-defined functionality;
- User experience during development;
- Flaws in documentation;
- Understanding of the working of the system (4.1.5: Maintainability);

Plugin within one hour

People with little programming experience should be able to write a plugin for a simple device within one day, as mentioned in the project objective in chapter 2. Since the user acceptance test was conducted with a HomeWizard developer, the expectation was to finish a plugin for the iKettle within half a day. Although the developer possessed no advanced knowledge of the Lua programming language and had no previous experience with Outsourcery, he managed to develop the iKettle plugin from scratch in a record time of only 45 minutes. Moreover, the overall reaction of the HomeWizard developer was “It works! And quite easy as well”, making the user acceptance test a huge success.

Developer tools

Most feedback received during the user acceptance test centered around the lack of proper developer tools. The process of testing a plugin during development is lengthy and cumbersome, and feedback from the system to the developer is minimal. Providing more extensive debug and networking logs during development would greatly improve developer experience and speed up plugin development even further.

Additional feedback

Two further feedback points are worth mentioning. The first one concerns loading plugins. Once the system installed a plugin, it loads this plugin from memory every time a new device is installed, thus not including changes made to the plugin since loading it. For developers, it would be more convenient if the system would reload the plugin into memory, including all changes. The second point of feedback was the side-note that module descriptions in the API should be ordered by functionality. For example, modules intended for discovery only, should be categorized as such, while communication modules should get their own category.

Developer experience

Another goal of the user acceptance test is to allow HomeWizard’s developers to get acquainted with Outsourcery. As the software system will be delivered to the client after this sprint, handing over knowledge of the system to HomeWizard’s developers is an integral part of closing the loop. This gives the developers who will continue developing, integrating and maintaining Outsourcery a head start on understanding the system and the motivations behind its design. The user acceptance test provides an ideal opportunity to introduce Outsourcery to HomeWizard’s developers.
8.3. SIG feedback

The second SIG feedback states that both the volume and maintainability of the code has increased since the first evaluation. SIG points out that the two areas they marked for improvement, Unit Size and Unit Complexity, have been greatly improved. Because of this, SIG concludes that their recommendations have been included in the development process. The code now has an overall score of four stars, which is a significant improvement over the three star score of the previous evaluation and qualifies above average. However, no possible points of improvement are mentioned. The complete SIG feedback has been included in appendix D.
Outsourcery is the final prototype of a plugin system for HomeWizard, handed over to the company for further development. To ascertain that the delivered product meets the expectations of HomeWizard and can be used in the future, this chapter provides an evaluation of Outsourcery in section 9.1. Outsourcery is a prototype exploring the potential and feasibility of a plugin-based home automation system. Therefore, it is not yet ready for deployment in a commercial product as distributed by HomeWizard. The future work required to turn Outsourcery into a complete and stable product usable in the HomeWizard hub is described in section 9.2. Recommendations to get Outsourcery to its fullest potential in the HomeWizard hub are given in section 9.3.

9.1. Evaluation of the product
To measure the success of the project, Outsourcery is evaluated against the design goals and requirements set out in chapter 4. A reflection on the design goals is given in section 9.1.1, followed by an evaluation of the requirements in section 9.1.2. Finally, an evaluation of the product against the project objective is given in section 9.1.3.

9.1.1. Reflecting on design goals
Outsourcery is designed to be flexible, simple, secure, and maintainable, while delivering high performance, as stated by the design goals in section 4.1. This section provides a reflection on these design goals. Reflections are supported by associated requirements, on which a reflection is provided in section 9.1.2.

Flexibility Outsourcery provides a flexible plugin system that is both extensible and modular. The Lua programming language provides a large amount of flexibility, despite its relatively small ecosystem. Moreover, any functionality that can be implemented with user mode C can be used by plugins in Outsourcery. Bundling commonly used functionality in modules allows plugin developers to quickly and easily switch between protocols. Plugins are thus highly flexible. Although the core of Outsourcery contains some architectural limitations, as described in section 8.1.1, the loose coupling between the core system and plugins provides enough flexibility to overcome these limitations.

Simplicity The user acceptance test described in section 8.2 suggests that plugins for single devices possess a high level of simplicity. Since the test was conducted with only one person, however, the achieved results may not be representative. Outsourcery provides a high level of simplicity through the use of modules for commonly used functionality and configuration statements rather than programmed functionality. However, the specific API that Outsourcery exposes to plugins must be learned by plugin developers. Moreover, developing complex functionality requires knowledge of the Lua programming language.
9.1. Evaluation of the product

Security Basic security is achieved by running each plugin in its own process, using kernel-level separation to provide plugin sandboxing. The flexibility of Outsourcery allows plugins to reach the maximum level of security offered by each device. However, no additional security measures are taken. Restricting the ability to run arbitrary C modules from plugins would improve security, but also greatly reduce the amount of flexibility. Further security measures should be carefully evaluated to achieve a balance between security and flexibility, discussed in more detail in section 9.2.

Performance Testing Outsourcery on a Raspberry Pi showed that the system runs without noticeable delay, as described in section 8.1.3. Moreover, running plugins in separate processes make Outsourcery both responsive and stable. Separate plugin processes, however, do require more resources than a single-process application. Nevertheless, Outsourcery is a light-weight system with a base memory footprint of under 7 MB (resident, of which under 2 MB private). Each currently developed plugin requires about 400 kB additional memory. Since the software does not limit the number of supported devices, the only limiting factor is the amount of available system resources, implying the software solution is scalable up to the 2 GB memory limit imposed by the current version of LuaJIT.

Maintainability The source code is fully documented and tested. The modularity of the system also improves maintainability. With a SIG score of four stars, the maintainability of the product scores ‘above average’.

9.1.2. Reflecting on requirements

The requirements, as stated in section 4.2, describe the required functionality for the software solution. Table 9.1 provides a short evaluation for each requirement, in order to indicate how the system satisfies the requirements.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Met?</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.</strong> The prototype must be capable of controlling at least one switch via TCP/IP.</td>
<td>Yes</td>
<td>The prototype can control the iKettle, the HomeWizard hub with a 433MHz switch, and the Philips Hue.</td>
</tr>
<tr>
<td><strong>2.</strong> It must be possible to implement the logic required to control devices over TCP/IP using plugins.</td>
<td>Yes</td>
<td>All communication logic that can be run in user mode can be run from plugins. Only privileged instructions, which should not be necessary for application-level communication over existing transport layers, cannot be executed.</td>
</tr>
<tr>
<td><strong>3.</strong> The system must expose the capabilities of device plugins, consisting of data that can be triggered on and actions to control devices.</td>
<td>Yes</td>
<td>Both triggers and actions can be specified in plugins. Triggers are monitored and updated on every assignment of a new value, whereas actions can be invoked from outside Outsourcery. A list of triggers and actions can be requested on a per-device basis.</td>
</tr>
<tr>
<td><strong>4.</strong> It must be possible to add and remove plugins while the system is running.</td>
<td>Yes</td>
<td>Plugins can be added and removed while the system is running.</td>
</tr>
<tr>
<td><strong>5.</strong> The system must run on a Linux-based operating system.</td>
<td>Yes</td>
<td>The system has been tested on four different Linux-based operating systems.</td>
</tr>
<tr>
<td><strong>6.</strong> The system must run on hardware equivalent with that of a Raspberry Pi.</td>
<td>Yes</td>
<td>The prototype has been successfully tested on a Raspberry Pi (see section 8.1.3).</td>
</tr>
<tr>
<td><strong>7.</strong> It should be possible to implement new application layer protocols using plugins, without having to change the system’s firmware.</td>
<td>Yes</td>
<td>New application layer protocols can be added to the system by writing modules implementing these protocols, as long as no privileged instructions are required.</td>
</tr>
</tbody>
</table>
### 9.2. Future work

Despite the success of this project as a whole, Outsourcery is still a prototype and as such not ready to be deployed in a commercial product. In order to turn Outsourcery into a complete and stable product usable in the HomeWizard hub, an overview of future work is given in this section.

#### Table 9.1: Evaluation of requirements.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Met?</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8. The system should allow reaching the maximum level of security offered by each device.</td>
<td>Yes</td>
<td>As long as the security protocols can be implemented on the application layer, or on a higher layer, security can be matched by writing a module that implements the security.</td>
</tr>
<tr>
<td>9. The system should provide barriers to limit interference between plugins.</td>
<td>Yes</td>
<td>The prototype separates plugins by creating a process for every plugin, making use of kernel-level sandboxing. Moreover, plugins are unaware of the existence of other plugins (see section 7.1).</td>
</tr>
<tr>
<td>10. Each module and each public function in the system should be documented</td>
<td>Yes</td>
<td>Each module and public function has been documented with LDoc documentation comments.</td>
</tr>
<tr>
<td>11. At least 85% of the code should be covered with unit tests.</td>
<td>Yes</td>
<td>The prototype is equipped with a testing suite which covers 93.08% of the code.</td>
</tr>
<tr>
<td>12. After development of the system, other engineers should be able to understand and maintain the code.</td>
<td>Likely</td>
<td>The code scores above average on maintainability, is fully documented and well tested. It is thus likely that other engineers will be able to understand and maintain the code within a reasonable amount of time. However, as the code has not been inspected by HomeWizard engineers, satisfaction of this requirement cannot be confirmed with certainty.</td>
</tr>
<tr>
<td>13. The architecture of the system should adhere to Software Engineering design principles, as described in Dr. A. Bacchelli’s Software Engineering Methods course.</td>
<td>Yes</td>
<td>Code analysis was performed by the Software Improvement Group. They awarded the prototype a score of four stars, which translates into “above average”.</td>
</tr>
</tbody>
</table>

**9.1.3. Evaluation of success criteria**

As described in chapter 2, the objective of this project was “to develop a prototype of a plugin system for HomeWizard. This plugin system should allow hardware manufacturers to add compatibility for TCP/IP-enabled devices to the HomeWizard hub. People with little programming experience should be able to develop a plugin for a simple device within one day.” Outsourcery, the final delivered prototype, is indeed a plugin system with which compatibility for TCP/IP-enabled devices can be added to the HomeWizard hub. The user acceptance test, described in section 8.2, suggests that development of a plugin for a single device is simple enough to be carried out within a day. Moreover, Outsourcery adheres to all design goals, although an expansion of security measures is necessary before the product can be adopted in a commercial product. All requirements have been satisfied, apart from requirement 12, which is likely to have been satisfied, but for which no conclusive confirmation can be given. Because the project objective, the design goals and all but one requirement have been met, the project can be regarded a success.
9.3. Recommendations

Security measures
Although kernel-level separation of plugins (section 7.1) provides basic security against malicious and faulty plugins, security should be an important concern in future work. Access of plugins to the filesystem should be limited, and a compromise between flexibility and security should be made.

Additional modules
Outsourcery is currently equipped with modules for performing IP range scanning, listening to UDP broadcasts, communicating via TCP and HTTP, and JSON encoding. While these modules already form a solid basis to develop plugins for many devices, more modules providing protocol implementations should be added to Outsourcery. Since Outsourcery is built to be extensible, additional modules can be added even after releasing the product.

Persistence and system recovery
The current version of Outsourcery is still a prototype, and one of the biggest missing features is device persistence. The current prototype does not save connected devices, so when the prototype is stopped, all devices are lost. Implementing functionality that stores connected devices is thus a necessary step to get Outsourcery ready for deployment. Once device persistence has been implemented, devices can be rediscovered after losing connection, and device-specific triggers and actions can be restored. This means that the system can recover from a system failure or temporary loss of power.

Package management
To simplify plugin development by giving developers access to a wide range of existing packages, as described in section 6.3, a package manager is required. Besides packages, a package manager can be used for plugin deployment as well. A package manager can be implemented either on a server or on the HomeWizard hub. Future work should determine an appropriate implementation of a package manager.

Interface
As Outsourcery only provides a backend for a home automation system, a user-friendly interface is required for the system to be useful as a home automation system. Therefore, a dynamic and flexible user interface for Outsourcery should be developed, automatically adapting to new plugins and devices as they are added. A follow-up project investigating the possibilities of a flexible user interface system to match the flexibility provided by Outsourcery is desirable future work.

Watchdog
Outsourcery utilizes multiple processes in order to prevent system crashes caused by faulty plugins. A watchdog could improve user experience by monitoring processes and act when necessary. For example, when a plugin is stuck in an endless loop, a watchdog can recover the process.

9.3. Recommendations
The following recommendations are devised to help HomeWizard use Outsourcery to its full potential. Applying these recommendations is not necessary for using Outsourcery, but we believe that, if followed, they will contribute to the success of Outsourcery and HomeWizard as a whole.

Outsourcery as core component
Outsourcery provides a solid core for a home automation system that offers a large amount of flexibility. It can easily be extended to support 433 MHz and 868 MHz devices, simplifying the process of adding compatibility for such devices. We therefore recommend HomeWizard to adopt Outsourcery as an integral part of the HomeWizard hub, rather than use it as a component attached to existing firmware. Using the largely standalone and extensible design of Outsourcery as a core component will provide the basis needed for a truly versatile home automation system.

Development platform and tools
Development of plugins can be greatly simplified when a proper development platform is available to provide developers with feedback on their plugins. A debugging tool that gives insight into detailed logs of system and network activity could prove very helpful during development. Static analysis of plugins
9.3. Recommendations

Recommendations to detect syntax errors or missing configuration fields would both simplify development and prevent faulty plugins. Moreover, a developer platform could assist in deployment of plugins and accessibility of documentation.

Involving customers
As Bianchi et al. [1] state, most of the current customer base of HomeWizard consists of enthusiasts interested in the technical possibilities of home automation. These enthusiasts often have the incentive to work around limitations they encounter. HomeWizard currently does not involve its customers in adding device compatibility, leaving their customers no option but to request HomeWizard to implement desired functionality. We recommend HomeWizard to accommodate an open, free and community-based platform, where customers can develop and share plugins for the HomeWizard hub. This strategy, although not directly leading to financial gain, will increase the number of supported devices, alleviate the workload of HomeWizard developers, and most importantly bind loyal customers to the HomeWizard platform.

We therefore recommend HomeWizard to enable customers to develop plugins by making Outsourcery’s API documentation and future development tools available to the public. Not providing an open platform to customers might lead to a loss of customers in the long term, as new competitors such as Homey\(^1\) and Nest\(^2\) provide a more attractive open platform for their customers.

\(^1\)Homey by Athom - http://www.athom.com
\(^2\)Nest - http://www.nest.com
Conclusion

Outsourcery, the prototype developed during this project, is a flexible and simple plugin system for the HomeWizard home automation hub. It allows manufacturers of TCP/IP-enabled home automation devices to easily make their devices compatible with the HomeWizard hub, thereby providing customers with a versatile yet affordable home automation solution.

The challenge to design and implement a system that is flexible enough to control hundreds of different devices, while being simple enough to be used by people with little programming experience, has led to a prototype using a modular plugin system utilizing the Lua programming language. The system allows plugin developers to use available components as building blocks using simple configuration statements, while also providing the flexibility needed to support complex behaviour.

After a two-week research phase, leading to the conclusion that the system needs to be incredibly flexible to support dozens of existing and future home automation standards, three successive prototypes were developed during scrum sprints of two weeks each. The final prototype, named Outsourcery, provides a solid core that, when developed further, enables HomeWizard to easily reach multi-vendor compliance and deliver a seamless yet affordable home automation experience.

As stated in the problem analysis, the objective of this project was “to develop a prototype of a plugin system for HomeWizard. This plugin system should allow hardware manufacturers to add compatibility for TCP/IP-enabled devices to the HomeWizard hub. People with little programming experience should be able to develop a plugin for a simple device within one day.” By creating a simple yet flexible plugin system, that can run on limited hardware, the project has been brought to a success.

Outsourcery, the final developed prototype, is handed over to HomeWizard for further development and maturing of the system. Being a flexible and maintainable system, Outsourcery forms a solid basis for a home automation system, ready for the future of the Internet of Things.
discovery = {
    protocol = 'ipscan',
    port = 2000,
    request = 'HELLOKETTLE\n',
    response = 'HELLOAPP\r',
    keepalive = true,
}

communication = {
    protocol = 'tcp'
}

function getStatus()
    send 'get sys status\n'
end

function setTemperature(temp)
    if temp < 80 then
        send 'set sys output 0x200\n'
    elseif temp < 90 then
        send 'set sys output 0x4000\n'
    elseif temp < 100 then
        send 'set sys output 0x2\n'
    else
        send 'set sys output 0x80\n'
    end
end

function setOn(on)
    if on then
        send 'set sys output 0x4\n'
    else
        send 'set sys output 0x0\n'
    end
end

function enableWarm(time)
    if time < 10 then
        send 'set sys output 0x8005\n'
    elseif time < 20 then
        send 'set sys output 0x8010\n'
    else
        send 'set sys output 0x8020\n'
    end
end
if not status.warm then
    send 'set sys output 0x8\n'
end

function disableWarm()
    if status.warm then
        send 'set sys output 0x8\n'
    end
end

function handleResponse(response, r)
    if r == '0x100' then status.temperature = 100
    elseif r == '0x90' then status.temperature = 90
    elseif r == '0x65' then status.temperature = 65
    elseif r == '0x11' then status.warm = true
    elseif r == '0x10' then status.warm = false
    elseif r == '0x5' then status.on = true
    elseif r == '0x0' then status.on = false
    elseif r == '0x8005' then status.warmlength = 5
    elseif r == '0x8010' then status.warmlength = 10
    elseif r == '0x8020' then status.warmlength = 20
    elseif r == '0x3' then status.reachedtemperature = true
    elseif r == '0x2' then status.problem = true
    elseif r == '0x1' then status.removed = true
    end
end
onReceive('sys status (0x%d+)/r', handleResponse)

function keyresponse(response, key)
    local num = key:byte()
    status.on = bit.band(num, 1) == 1
    status.warm = bit.band(num, 2) == 2
    status.temperature = bit.band(num, 32) == 32 and 100
    or bit.band(num, 16) == 16 and 90
    or bit.band(num, 8) == 8 and 80
    or bit.band(num, 4) == 4 and 65
    or 0
    return true
end
onReceive('sys status key=(.)\r', keyresponse)

trigger 'on' 'device is %b'
trigger 'warm' 'device keeps water warm %b'
trigger 'temperature' 'temperature is set to %d degrees'
trigger 'warmlength' 'warm length is set to %d minutes'
trigger 'reachedtemperature' 'temperature is reached'
trigger 'problem' 'device has a problem'
trigger 'removed' 'device is removed while on'

action 'setTemperature' 'set temperature to %d degrees'
action 'enableWarm' 'warm for %d minutes'
action 'disableWarm' 'disable warm mode'
action 'setOn' 'turn device %b'

Listing A.1: Fully functional implementation of the iKettle plugin.
B.1. HomeWizard.switch plugin

```plaintext
-- The gateway discovery protocol is a build-in mechanism that enables a plugin to
query gateways if a compatible device is connected to them.
discovery = {
  protocol = 'gateway',
  gateways = 'HomeWizard',
  category = 'switches',
  type = 'switch',
  prefix = 'sw'
}

-- The gateway communication protocol is a build-in mechanism that enables a
plugin to use a gateway plugin to handle communication.
communication = {
  protocol = 'gateway'
}

function setOn(state)
  local res
  if state then
    res = send 'on'
  else
    res = send 'off'
  end
  status.on = res and res.status == 'ok' and state or false
end

action 'setState' 'set switch %b'
trigger 'on' 'is turned %b'
```

Listing B.1: Implementation of the HomeWizard.switch plugin used to control switches via a HomeWizard hub.
B.2. HomeWizard gateway plugin

```lua
B.2. HomeWizard gateway plugin

discovery = {
    protocol = 'broadcast',
    port = 55555,
    pattern = 'HomeWizard',
    timeout = 10
}

communication = {
    protocol = 'http',
    encoding = 'json',
    prefix = 'password',
}

function getStatus()
    local res = send 'get-status'
    status.error = res and res.status ~= 'ok'
end

function setPassword(password)
    device.password = password
    communication.prefix = password
end

-- The sendTo function specifies how the gateway should communicate with its children.
function sendTo(child, command)
    return send(child.prefix .. '/' .. child.id .. '/' .. command)
end

-- The discoverChildren function specifies how the gateway should discover if it has devices connected to it. This function is called when the plugin of a connected device tries to discover its devices.
function discoverChildren(arg)
    local res = send 'get-sensors'
    for k, v in pairs(res.response[arg.category]) do
        if not arg.type or v.type == arg.type then
            local child = createChild(v.name .. '_' .. v.type .. '_' .. v.id)
            child.id = v.id
            child.prefix = arg.prefix
        end
    end
end

action 'getStatus' 'Get the status of all sensors and devices connected to the HomeWizard.'
action 'setPassword' 'set the Password of the HomeWizard'
trigger 'error' 'has encountered an error'
```

Listing B.2: Fully functional implementation of the HomeWizard plugin. HomeWizard devices broadcast a UDP datagram on port 55555 every 8 seconds.
Tutorial: Developing a simple plugin

This tutorial walks through creating a simple plugin for the HomeWizard hub. Examples of existing plugins can be found in the plugins directory.

Lua scripting

Plugins for the HomeWizard hub are created using the Lua scripting language. Although basic programming experience is sufficient for most plugins, some knowledge of Lua may come in helpful. Lua basics are covered at http://luatut.com. The full Lua manual is located at http://www.lua.org/manual/5.1/.

Creating an empty plugin

Plugins are simple text files named `<name>.lua`, where `<name>` should be replaced by the name of your plugin. It is generally a good idea to name a plugin after the device it supports. The plugin should be saved in the plugins directory. Once this is done, actual implementation can start.

Adding functionality

A plugin requires a few components in order to work correctly;

1. A **discovery** method, to discover the device on the network.
2. A **communication** method, for communication with the device.
3. **Actions & triggers**, which define the device capabilities.

1. The discovery method

The discovery method must be defined to be able to discover devices on the network. In the simplest form, the discovery method is a table named `discovery` with a field named `protocol`. The `protocol` field should contain the name of a discovery protocol between quotes, such as "ipscan". Additionally, protocol-specific parameters may be added to the `discovery` table. Consult the documentation of the used protocol for possible parameters. Listing C.2 contains an example of a discovery definition, with `port` being a parameter specific to the ipscan protocol.

```lua
discovery = {
    protocol = "ipscan",
    port = 2000
}
```

Listing C.1: Defining the discovery method.

The discovery method will be performed every time a user tries to add devices to the HomeWizard. If new devices are discovered, they will be added to the HomeWizard and ready for use.
Advanced functionality

If functionality is required that is not provided by an existing module, you can implement your own discovery function. Instead of creating a table named `discovery`, create a function named `discovery` that implements the device discovery functionality. For every device that is found, call the function `createDevice(<unique_id>)`, with `<unique_id>` substituted for a unique device identifier. Alternatively, you can write a re-usable discovery module containing the discovery function.

2. The communication method

The communication method must be defined to be able to send and receive messages to/from devices. In the simplest form, the communication method is defined using a table with a protocol field. The table must be named `communication`. The protocol field should contain the name of a communication protocol between quotes, such as “http”. Like the discovery table, additional protocol-specific parameters may be added to the communication table. Listing C.2 contains an example of a communication definition, with `path` and `method` being parameters specific to the `http` protocol.

```
communication = {
  protocol = "http",
  path = "api",
  method = "GET"
}
```

Listing C.2: Adding the communication method.

With the communication method defined, you can send data to the device with the `send` function, and receive data with the `receive` function. Actually, `send` is a short name for `<protocol>.send`, where `<protocol>` should be replaced by the name of the communication protocol. When using the `http` protocol, for instance, send is a short name for `http.send`. With the example in listing C.2, `send('on')` would send an HTTP GET request to `http://<device-ip>/api/on` and return the received response. Likewise, receive is a shorthand for `<protocol>.receive`.

Apart from send and receive, protocols may define more operations. The `http` protocol, for instance, defines `http.get`, `http.post`, `http.put` and `http.delete`. For the usage of send, receive, and possibly other protocol-specific functions, consult the documentation of the used protocol.

Some protocols may define a function called `onReceive`. This function can be used to specify which function should be called when data is received. Listing C.3 contains an example of using `onReceive` function with the `tcp` protocol. The `onReceive` function in the `tcp` protocol expects a `pattern` as first argument. If incoming data matches that pattern, the specified function is called with the received data. For instance, if the data `Pattern 7` is received in listing C.3, the function `handle_receive` is called with `Pattern 7` as first argument, and 7 as second argument. For more information on patterns, the chapter on patterns in Programming in Lua is an excellent resource (find it online at [http://www.lua.org/pil/20.2.html](http://www.lua.org/pil/20.2.html)).

```
communication = {
  protocol = "tcp"
}

function handle_receive(data, pattern_number)
  if pattern_number == "7" then
    status.lucky = "Lucky number 7!"
  end
end

onReceive("Pattern (%d+)", handle_receive)
```

Listing C.3: Using the `onReceive` function.

Advanced functionality

If functionality is required that is not provided by existing modules, the send and receive functions can be overwritten within the plugin. Alternatively, you may consider writing a new communication
module implementing the send and receive functions.

3. Actions & Triggers
The next step is to implement actual device functionality. Functionality consists of actions and triggers. Actions define what users can do to control the device, e.g. turning a switch on and off. Triggers describe status changes of the device of which the user can be notified, e.g. a smoke detector detecting smoke.

Actions
Actions should be specified in the form action "function_name" "action description". function_name should be the name of a function implementing the action, whereas action description should provide a human-readable description of the action. Listing C.4 contains an example of an action named "turnOn" with the description "turn the device on". The function turnOn contains the actual implementation of the action, sending an HTTP GET request to /api/on if the function's argument is true, and an HTTP GET request to /api/off otherwise. Note that the actual communication is handled by the specified http protocol.

```
communication = {
  protocol = "ipscan",
  port = 2000
}

discovery = {
  protocol = "http",
  path = "api",
  method = "GET"
}

function turnOn(argument)
  if argument == true then
    send("on")
  else
    send("off")
  end
end

action "turnOn" "turn the device on"
```

Listing C.4: Adding device functionality.

In the example, device functionality is added in two steps. First, a device function named turnOn is declared. In this function, a control instruction is sent to the device. Secondly, the function has to be exposed as an action. This is done in the last line of the example.

Triggers
Triggers should be specified much like actions, in the form trigger "trigger_name" "trigger description". trigger_name should be the name of a variable to watch, whereas trigger description should provide a human-readable description of the trigger. A trigger can be updated by assigning a value to status.<trigger_name>. The code status.on = true, for instance, updates the trigger with the name on to the value true. This can subsequently be used by the HomeWizard to notify the user, or perform another action. Listing C.5 illustrates the use of triggers.

```
communication = {
  protocol = "ipscan",
  port = 2000
}

discovery = {
  protocol = "http",
  path = "api",
  method = "GET"
}
```

Listing C.5: Illustrates the use of triggers.
method = "GET"
}

function turnOn(argument)
    local reply
    if argument == true then
        reply = send("on")
    else
        reply = send("off")
    end
    if reply == "ON" then
        status.on = true
    else
        status.on = false
    end
end

action "turnOn" "turn the device on"
trigger "on" "the device is turned on"

Listing C.5: Adding device functionality.

In this example, the response to the HTTP GET request is used to update the trigger named on. Some plugins may use the onReceive function to update triggers, such as in the example in listing C.3, where the lucky trigger is set to "Lucky number 7!".

Implementation details

During this tutorial, functions like send and onReceive were used. The way these functions were used are in no way the only way these functions could behave. The documentation of the communication module in use should always be checked in order to use these functions correctly.

Finally: running the plugin

Now that the plugin contains discovery and communication details and exposes actions and triggers, it can be used to control devices.

To run the plugin system, start a terminal and run

$ luajit main.lua

Now connect to the plugin system by running the client from a second terminal:

$ luajit client.lua

Commands to the plugin system can be entered in the client. The following commands install a plugin and execute an action. The ‘1’ on the second line indicates the action named turnOn should be performed on the device with index 1. The ‘true’ at the end of the second line passes the argument true to the function implementing the action turnOn.

> install 'pluginName'
> 1 doAction 'turnOn' true
**D.1. First SIG feedback**

De code van het systeem scoort 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Unit Size en Unit Complexity.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Het is echter onjuist om te verwachten dat een methode simpel te begrijpen is, als het probleem dat een methode oplost moeilijk te begrijpen is. Als een methode veel beslispunten heeft wordt hij vanzelf moeilijk te begrijpen. Daarnaast moet je voor elk beslispunt een testgeval schrijven, en onze ervaring is dat dit bij een te groot aantal beslispunten niet meer gebeurd. In jullie project is iKettle.lua een goed voorbeeld. Onderaan staat een grote if/else die vrij moeilijk leesbaar is. Je kunt de implementatie versimpelen door dit mechanisme op een andere manier op te schrijven. Als richtlijn voor kun je 5 beslispunten per functie gebruiken, als je daar overeen gaat moet je gaan nadenken of het probleem niet op een andere manier opgelost kan worden.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de langere methoden ook naar voren als de meest complexe methoden, waardoor het oplossen van het eerste probleem ook dit probleem zal verhelpen.

Tot slot hulde voor het schrijven van unit tests. Bij andere Lua-systemen die tegenkomen is dat zeker geen automatisme, dus het is goed om te zien dat jullie test-driven werken serieus nemen. Hopelijk blijft de hoeveelheid testcode stijgen op het moment dat jullie project verder groeit.

**D.2. Second SIG feedback**

In de tweede upload zien we dat zowel het codevolume als de score voor onderhoudbaarheid zijn gestegen. Jullie zijn aanzienlijk omhoog gegaan, en scoren met 4 sterren nu bovengemiddeld. De stijging is veroorzaakt door een grote stijging op de aspecten Unit Size en Unit Complexity, die tijdens de analyse van de eerste upload als verbeterpunt werden aangemerkt. Daarnaast is de hoeveelheid testcode verder gestegen. Zoals eerder opgemerkt is dat voor Lua-ontwikkelaars meestal geen automatisme, dus complimenten daarvoor.

Uit deze observaties kunnen we concluderen dat alle aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.
Original project description

Project description
Steeds meer apparaten krijgen een verbinding met internet. Deze apparaten kunnen daardoor geautomatiseerd worden aangestuurd. Het compatibel maken van apparaten met een bestaande automatiseringsdienst vergt veel tijd en werk. Een adapter-framework kan daarbij helpen. De opdracht is om (een prototype van) een framework te ontwikkelen, dat hardwarefabrikanten in staat stelt zelf eenvoudig hun hardware te koppelen.

Company description
HomeWizard B.V. is een jong dynamisch bedrijf dat zich richt op het automatiseren van bestaande (veelal) consumenten producten. Door een centrale gateway in het huis die kan communiceren met WiFi, 433 MHz en 868 MHz zijn vele producten aan te sturen door middel van onze apps.
Title of the project: Outsourcery: simply extending home automation using plugins
Name of the client organization: HomeWizard B.V.
Date of the final presentation: June 26, 2015

Description
HomeWizard B.V. is looking to expand the range of compatible TCP/IP devices for its home automation hub. To this end, the company wants to use third-party plugins that control these devices. To allow even people with little programming experience to make devices compatible with the HomeWizard hub, these plugins have to be easy to create.

These demands pose an interesting challenge: to create a system that is flexible enough to control hundreds of different devices, while being simple enough to be used by people with little programming experience. During the research phase of the project, existing home automation solutions were studied, as well as home automation techniques. The clients current position in the market was also researched to get a better understanding of the clients needs.

During the implementation of the product, scrum was applied to manage the project. Prototypes were created in sprints of two weeks.

A final prototype, called Outsourcery, is delivered at the end of the project. Outsourcery is a flexible home automation solution, facilitating the use of plugins to extend compatibility with both devices and network protocols. Outsourcery has been tested using unit tests and manual testing. An acceptance test determined that the creation of plugins is relatively easy.

Outsourcery is a good start towards a universal home automation solution. However, it has still some way to go before it reaches maturity. Some required steps have been detailed in the report. Furthermore, some recommendations are done regarding the development of a plugin developer platform and making said platform available to customers.

Team members
Gijs Wassenaar  Role: Product owner, Developer
Leon Helsloot  Role: Lead developer, Quality assurance
Thijs Brands  Role: Scrum master, Developer

Contact
Client: Erwin Marges, Co-Founder HomeWizard B.V.
Coach: J.A. Pouwelse, Parallel and Distributed Systems, TU Delft
Contact person: Gijs Wassenaar, mail@gijswassenaar.nl

The final report for this project can be found at: http://repository.tudelft.nl
Bibliography


