Term project

Intelligent, Learning System
a new ABI System built on the Open Services Gateway initiative

Stephan Kei Nufer
<snufer@ini.phys.ethz.ch>

Mathias Buehlmann
<mbuehlma@ini.phys.ethz.ch>

Advisors
Prof. Dr. Rodney Douglas, Institute of Neuroinformatics, ETH/University Zurich
Prof. Dr. Josef Joller, University of Applied Sciences Rapperswil
Tobi Delbruck, Group Leader, Institute of Neuroinformatics, ETH/University Zurich

A cooperation between

Computer Science Department
University of Applied Science Rapperswil
Oberseestrasse 10
8640 Rapperswil, Switzerland
http://www.hsr.ch

Institute of Neuroinformatics
University and ETH Zurich
Winterthurstrasse 190
8057 Zurich, Switzerland
http://www.ini.unizh.ch

Compiled: February 7, 2006

Preface

by Nufer Stephan Kei and Buehlmann Mathias

This article is the main part of our term project and chiefly addresses ABI System related aspects such as its design, architecture and most importantly a complete description about how the system has been implemented using OSGi bundles ([OSG]).
Next to this document there are two related documents which are also part of this term project.
These are:

- The RBC Protocol Specification ([NB05b])
- The RBC API ([NB05c])

In particular the RBC API will play a central role in our upcoming diploma thesis.

Acknowledgment

Special thanks goes to our professor Prof. Dr. Josef M. Joller who had given us the opportunity to accomplish an important part of the ABI System at the Institute of Neuroinformatics(INI) as a semester project assignment.

Further credits goes to:

- Tobi Delbruck for the implementation of the enhanced PC Presence Sensor
- Our predecessors for providing us helpful information

Zurich, Institute of Neuroinformatics(INI), Switzerland, July 2005

Stephan Kei Nufer and Mathias Buehlmann

Abstract The presented term project is aimed at developing a new ABI System that has partly been adapted from our predecessors which in turn needed to be completely refactored in order to benefit from any advantage that an OSGi Framework might bring along in the sense that bundles can now implement new ABI System features completely independent of each other.
We achieve this by exposing well defined interfaces which are located in a central bundle called core. Additionally we introduce a new concept called *Property Concept* that considers that several different devices may coexist within one building. Each device type can hereby provide its own idiosyncrasies in form of properties and thus facilitating any generic access to its features.

To give a practical example of how this generic concept can be used we integrated a new custom bus called falcon bus that incorporates new devices which can be assessed by its dedicated bus that addresses them. According to the bus concept that has been introduced by our predecessors each device-set such as the falcon device-set needs to implement a separate bus. The reason why such a distinction has been made is because the concept presumes that each device-set is controlled by some kind of controller. In the lon bus this happens to be a gateway called LNS Server and in the newly integrated falcon bus this is a gateway called *Falcon Communicator* that represents a similar device that communicates with its sensors and actuators using wireless instead of the lon technology.

Further we present a distributed client application solution that simplifies any future development of custom client applications associated with the ABI System in the sense that it provides a compact application layer standard and reference implementation that can be used to remotely control a building (RBC). Thus improving any development practices such as complex integration testing, debugging and maintenance remarkably. Herewith we strike a familiar path that has already been applied in a previously distributed ABI System ([TZ03a], [TZ03b], [RS02]). In contrast to the latter system though we introduce two different standards that define the system boundary. The first standard introduces a *Remote Building Control Protocol* (RBC) that standardizes the communication between any distributed client application and the ABI System ([NB05b]). The second standard has been implemented by the previously mentioned reference implementation that replicates the entire ABI System that can be used by any distributed client application ([NB05c]).

Furthermore the application layer standard ([NB05c]) can now be extended by providing a bundle specific implementation instead of a remote one and hereby allowing any distributed client application to be turned into either a separate bundle or a distributed client application.

In order to prove the concepts we developed a variety of distributed client applications that make use of the remote API. Among other GUI applications we realized an application that is capable of managing the ABI System in a distributed kind of fashion.

Finally we provide additional documentation materials that clearly outline distinct agreements that are required to be adhered. Herewith we should achieve a system that can be kept low coupled, stable, easy to maintain, expandable, easy to get acquainted with and most importantly we can reduce the point of failures to a minimum.

Table of Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Retrospection

Our first task was to develop an intelligent learning system that should be capable of controlling an illumination system based on wireless effectors and sensors.
The motivation took place upon a cooperation between the University of Applied Sciences Rapperswil and a company called Feller AG that has been doing related research work in this field quite a while ago. Additionally they provided us with the necessary equipments such as the wireless devices (light effectors, presence and daylight sensors).

Our Professor, Prof. Dr. Josef Joller indicated that it would be advisable to perform such kind of researches in an environment that has already been configured and aligned for such kind of purposes. In an incitement he pointed out that an ABI System has recently been developed that might help to solve at least a chunk of our task since it is related and secondly the latter introduced abstract bus concept (See: [BG04a]) might be even possible to be adapted in our context as well.

1.2 History

Many research work is concerning about the integration of autonomous intelligence into a building in our everyday life. Many attempts are trying to interact, improve user comfort and provide security in modern working and living environments. Here at the Institute of Neuroinformatics(INI) a couple of projects have been performed that mostly were concerned in developing a System that should act with the environment through common devices like lights, window blinds etc. and senses from illumination-, temperature-, radiation-, daylight-sensors and presence detectors. As a follow up each of the projects also tried to solve if a building can be adaptively act intelligent. Intelligent in the sense that a building is changing in nature from static structures of bricks and mortar to dynamic work and living environments that actively support and assist their inhabitants ([RJD04]).
These new buildings are expected to behave intelligently. In order to do performance analysis of an intelligent system in a real working and living environment, they need access to all sensors and effectors within rooms, floors or even a whole building. All projects have been conducted at the Institute of Neuroinformatics (INI) where they had the opportunity to interact with a given field bus (LON) that is capable to communicate with all its connected sensors and effectors (See [TZ03a]).

1.3 Adaptive Building Intelligence (ABI)

Before digging into the OSGi based ABI System solution we briefly recap the actual state of affairs since there were quite a few and might help to catch onto.
Basically two adaptive building intelligence systems (ABI Systems) have now been developed by several term projects and diploma thesis. Evidentially each system turned out to be very unstable and sometimes took unexpected actions and this after a couple of months of being tested. Neither system barely was able to survive a month without having crashed. Evidentially the inhabitants that were using the OSGi based solution were not satisfied with it.

A building is a very complex system, since it behaves from a computational point of view completely nondeterministic. Raphael Zwicker, Jonas Trindler, Ueli Rutishauser and Alain Schaefer have developed a multi-agent-system that they called DAI (See: [TZ03a], [TZ03b], [RS02]). DAI stands for Distributed Artificial Intelligence and basically represents an ABI System that is constructed out of agents, each of which act independently and communicate indirectly with others about their goals and actions they take. Every agent has its own field of responsibility, but to achieve the overall goal of controlling a building they have to collaborate.

To retrospect the recently developed ABI System: The ABI System based on OSGi has been developed from our predecessor, Patrik Brunner and Simon Gassmann (See: [BG04a]) and has been structured as a set of agents that are software components that are mostly independent and autonomous. Each of these agents pursue their own goals and cooperate with other agents whenever it is necessary. In the OSGi Framework, agents have to be compared with services. Software components hereby represent a so called OSGi Bundle that consists of a set of services.
In contrast to the DAI-based ABI System, the OSGi-based solution has rather been characterized as a centralized Software System solution. This means that most of the things rather concentrate in a single point of failure whereas the DAI-based solution has been constructed as a set of distributed agents and might need to consider multiple points of failures.

In outlook to our diploma thesis in October 2005 we try to combine our research work and adapt the latter ABI system based on OSGi. Hence it is necessary to introduce their work to understand what the open issues and needs are which have to be analyzed and satisfied first, before we can define and illustrate our objectives (See section: 2.3) of our work.

1.4 Terms and definitions

For the sake of clarity we use the term RBC API to refer to the remote implementation of the API when not explicitly stated otherwise (See [NB05c]).
Other terms and definitions are to be looked-up in the glossary (See appendix section: B)

1.5 Preconditions

This document presumes that that the reader is already familiar with the term OSGi and ABI and ABI System. We also assume that the reader has at least read the previous term project ([BG04b]) and the diploma thesis of Patrik Brunner & Simon Gassmann ([BG04a]).
Therefore we don't recap and or provide any details about the OSGi Framework nor we cover any fundamental ABI System aspects in here as well. Concretely speaking this means that terms such as: bundles, Services, ServiceReferences, etc should not be a foreign word to you. For more details please consult ([OSG]) or previous term projects and diploma thesis.

1.6 Document structure

This document is structured into five parts: An Introduction part: I, an Analysis part: II, an Architecture part: III, a Design and Implementation part IV and an Appendix part that concludes of a Glossary and a Bibliography chapter (See: V).

The first Introduction part basically summarizes the state of affairs and retrospects previously conducted projects. The next part then analyzes the previously developed ABI System that has been completely implemented using an OSGi Framework and outlines the issues that need to be addressed. In a sub-chapter (See chapter: 3) we analyze the ABI System goals that we want to achieve.

In the following architecture part (See part: III) we then present our concrete solution to each of the addressed proposition stated in chapter: 3. We illustrate which bundles have to be implemented and show how they collaborate with each other in a low coupled way.

The design and implementation part (See part: IV) is then rather digging into the concrete realization of each bundle in a comprehensible way in the sense that it provides a mixture between descriptive text, UML diagrams and code snippets.

Finally we close up the paper with an appendix part that may provide additional information such as certain protocol definitions.

We leave out any discussion and future work chapters since they have already been addresses by the analysis section (See section: 3.10) as well as within the accompanying RBC API documentation [NB05c] that among other things forecasts what needs to be done in the upcoming diploma thesis.

Part II

Analysis

Chapter 2

Analysis of the previous ABI System

2.1 ABI System issues

We recall that our task is to incorporate the wireless devices (so called Falcons) into the existing ABI System based on OSGi.

As we were getting acquainted with the ABI System we noticed that most parts of the system were not developed to be adaptable and neither they were stable and independent of each other. We have been realizing this as we were about to fit the wireless devices into the system and also when planing the development of a graphical user interface application that actually should manage and configure the ABI System (See section: 2.3).

When facing the fact that the ABI System has been fully implemented using bundles one might think that the adaptability as well as the extendability are greatly preserved and ensured. Hereby the bundles are comparably related to applications rather then regular packages as most applications have been implemented with. Unfortunately we've noticed that momentarily this isn't the case. Imagine having the task of assembling a bunch of separate applications in a way that they should interact with each other but you can't really see the interaction going on between the required applications. Concretely speaking, when unrolling the dependencies in the ABI System on can see the big mess we have inside. Especially with the usage of the `Wireadmin` (See section: 2.1.1) that in the current context produces rather more hard "wired" dependencies between the services with the result that barely anyone is capable of keeping track of the entire system anymore.
The flip side to this coin is that bundles in this context are praised to be developed independently and still require each other in an invisible way.
Figure how such as system can possibly being maintained by a semi-yearly shift of the developers.

2.1.1 Wire Admin Issues

The **Wire Admin** represents an administrative service that is used to control a wiring topology in the OSGi Service Platform. It is intended to be used by user interfaces or management programs that control the wiring of services in an OSGi Service Platform. The Wire Admin service plays a crucial role in minimizing the amount of context-specific knowledge required by bundles when used in a large array of configurations. The Wire Admin service fulfills this role by dynamically wiring services together. Bundles participate in this wiring process by registering services that produce or consume data ([OSG]).
The Wire Admin service wires the services that produce data to services which consume data. Hereby producers and consumers get interconnected in a loosely coupled way called a wire. Generally speaking producers are services that generate data intended to be used by consumers. In order to receive any data generated by a producer corresponding services must be implemented that conform to those roles. When the

services have once been registered to the OSGi Framework, the `Wireadmin` is able to create a wire between the services and allow the exchange to be taken place, provided that both parties provide the same kind of flavour that should logically correspond to its requirements.
Basically one can say that the concept of the `Wireadmin` is quite similar to that one provided by the common observer patter although additional features have been added such as the capability to persist wires that can be re-initialized upon a crash or a bundle stop.

Apparently the concept of the `Wireadmin` must have been misunderstood since the flavours did not correspond to any standard format and hence was causing heavy hard "wired" dependencies between the services and its bundles. Hard "wired" in the sense that not every presence service can simply be seen as a generic presence service that only provides a status that can be either true or false. Consider a specific presence service that might provide additional features such as the capability to query a battery status or may provide a frequently sent daylight value.
For such devices in particular it is mandatory to be aware of all provided features. For instance a battery driven presence device might send its battery status when being low-powered and might give an extra hint about its weak existence and hence would allow an application software to consider this option while keeping track of it.

Specifically speaking a falcon presence-daylight device provides a combination of different sensors (daylight and presence) within one entity. The challenge we got confronted with was how to distinguish between the different data that must be recognized by a `Consumer` when not considering the option of parsing through each different possible string that has been provided by each concrete device implementation. Evidentially no such solution exists since we cannot derive from two different interfaces that prescribe the `Consumer` marker interface.

2.1.2 Other Issues

Other issues were that static configurations were depending on the Lon Bundle ([BG04a]) in such an extent that it was practically impossible for us to adapt this bundle and hence needed to postpone the support of the lon bus into the diploma thesis and only consider the realization of the falcon bus (See section: 3) within this term project.
Furthermore the ABI Area Bundle was totally confounded with an intelligent related part which should frankly speaking be free and independent of any of such.
There would be a bunch of more issues that need to be addressed and refactored since nobody would be capable to add any features to this ABI System and since it causes problems nobody can see through it.
One has to note though that our predecessors were in a big rush since they did everything from scratch and even using a new technology ([OSG]) that needed to be learned first. The most severe part though was the misleading usage of the `Wireadmin` (See section: 2.1.1).

2.2 OSGi Discussion

Although OSGi has emerged into a very powerful and adaptive system framework, it also has its liabilities. Those negative forces need to be circumvented in a way that it does not effect the hot-plugging capabilities. In order to recall why it is worth to apply an OSGi Framework in an adaptive building intelligence system (ABI System) we itemize some advantages first.

Advantages:

- The OSGi service platform specification delivers an open, common architecture for service providers, developers, software vendors, gateway operators and equipment vendors to develop, deploy and manage services in a coordinated fashion ([OSG]).
- It enables an entirely new category of smart devices due to its flexible and managed deployment of

services.

- OSGi reduces the incompatibility problems of legacy systems. The key is the common service interfaces within the OSGi service framework
- The platform designed by OSGi is open and scalable for the delivery of new services. This platform enables manufacturers and service providers to bridge the gap between all common standards and vertical markets and to communicate with the end consumer via one single service gateway platform specification [cellular.co.za]

There are also a couple of liabilities or violations that need to be prevented when considering an OSGi Framework. Some of them really need to be addressed to accomplish our objectives (See section: 2.3).

- **Development:** The art of programming is partly changed in the way that it might restrict the developers in a form of a schema which might impact the developers design skills in a negative way.
- **GUI Problem:** Evidentially a graphical user interface application is not meant to be developed as a separate bundle inside an OSGi Framework since a window closing would cause the entire system to be shut down. So we have to think of a client server model (See section: 3.5) rather then a bundle solution. Frankly speaking this is even a more beneficial approach as we identify later: (See section: 3.5 and section: 3.9 for more details).
- **Deployment:** Building OSGi conform bundles can be quite tedious when done by hand or with ant solutions.
- **Maintenance:** A system that is composed out of bundles must be carefully designed in order to prevent a system to be difficult in maintenance.
- **Dependencies:** Each bundle must comply to a standard in order not to cause "hard-wired" dependencies. The interaction between the bundles must be adaptive and low coupled. A bundle should not require to possess any conditional knowledge of other bundles which do not conform to a generic interface or principle.
- **Initial skill adaptation training:** When solving the forces stated above we can enhance the time that other developers would need to get acquainted with the System.

2.3 Our Objectives

The actual term project intends to refactor and stabilize the design of the existing ABI System which has been developed in the diploma thesis of Patrick Brunner & Simon Gassmann ([BG04a]). The negative forces stated above (See section: 2.2) need to be circumvented in a way that it does not effect the hot-plugging capabilities.

Further tasks are:

- To develop a new bus that includes the support of wireless devices (so called Falcons).
- The development of Graphical Configuration Utilities
- Management of the system
- Provide tutorials and howto's that facilitates further development practices.

Chapter 3

Analysis of the new ABI System

3.1 Introduction

In this chapter we chiefly introduce how we counteract the issues that have been identified in section: 2.2. Further we cover some additional topics such as some details about the falcon devices (See section: 3.2) and discuss some future aspects since they might have an impact in our architecture and the design as well.
We also cover some fundamental aspects that have been covered in section: 3.9 that might be worth to take a look at.
Section: 3.6 additionally briefly outlines that we might need to consider advanced presence detectors since evidentially the ordinary ones are not well suited for most environments (See also: [TZ03b]).

3.2 The Falcon devices

According to the *bus concept* that has been introduced by our predecessors ([BG04a]) each device-set such as the falcon device-set needs to implement a separate bus. The reason why such a distinction has been made is because the concept presumes that each device-set is controlled by some kind of controller. In the lon bus this happens to be a gateway called *LNS Server*. The *Falcon Communicator* (See section: 3.2.1) as you will see represents a similar device that communicates with its devices using wireless instead of the lon technology.
Hence one can simply say that each bus basically implements a proxy that replicates such a controlling entity in the sense that it should provide a way to *connect* and *disconnect* the bus and furthermore implement methods that allow any device to be *registered* and *deregistered* to respectively from the bus. In summary one can say that each bus provides some kind of door to its devices.

Section: 3.2.1, 3.2.2 and 3.2.3 describe the device capabilities and outlines some undocumented features.
For more technical documents consider the following papers: ([EAS04a], [EAS04c], [EAS04b], [AG05], [VIS99], [TAO03], [TAO01], [OPT05], [WIR05]).

In order to get acquainted with the devices some tools have been developed by Urs Schlegel (See: [FAL]).

Some fundamental research ideas and simulations might also be of interest: (See: [Sch04]).

Figure: 3.1 illustrate the equipments that have been used in the falcon system.

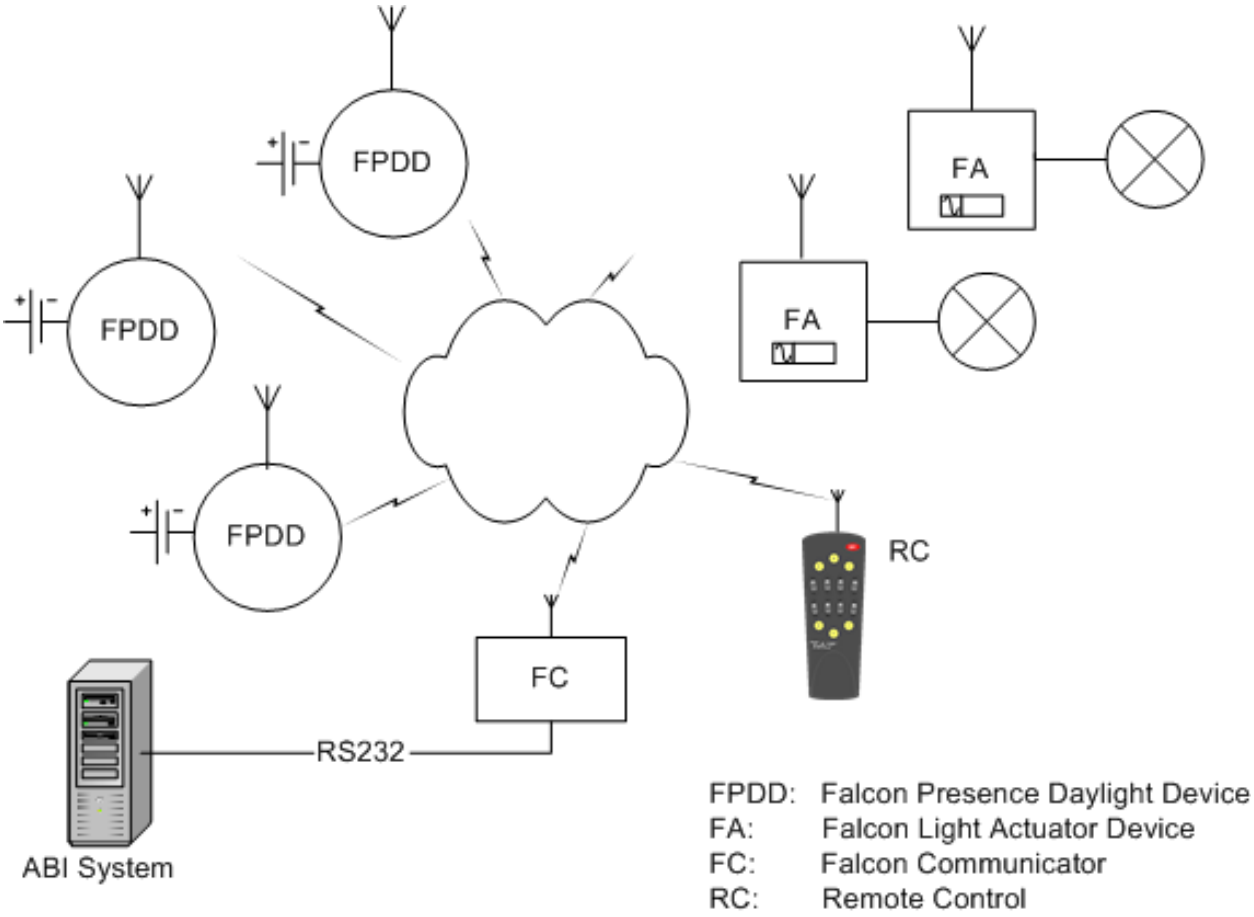


Figure 3.1: Falcon System

3.2.1 The Falcon Communicator

Falcon devices refer to wireless devices that are interconnected with a so called *Falcon Communicator* that represents a receiver that serves as a central communication station between each interconnected falcon device and a PC-Client. In order to communicate to the devices the Falcon Communicator provides a RS232 connector that can be used to either send or receive commands.
Technically speaking the Falcon Communicator concludes of an Easy-Radio ER900TS Transmitter and an ER900RS Receiver. The ER900TRS transceiver incorporates an "Easy-Radio" technology that provides high performance that can control radio devices that can transfer data over a range of up to 250 meters Line Of Sight (LOS).

3.2.2 The Falcon Presence-Daylight Device

The presence and the daylight is measured by a wireless battery powered presence detector and an accompanying daylight sensor.
Unlike the separate lon ambient daylight and the lon presence sensor, the Falcon presence daylight provides a combination of the two. At the moment the device is programmed to deliver new daylight and presence information when either measuring a light delta of 10% to the previous one or when detecting a presence change. Additionally this devices provides keep-alive messages which are broadcasted hourly.

It's important to note that this device generally does not support any command-set that would allow to concretely query for any specified device information such as the daylight or the presence value. The reason of this is that the device is usually acting in passive-mode since its trying to save energy whenever possible.

This is due to the fact that this device is battery powered and hence would consume to much of power when serving twenty-four-seven.

However there is one command that can be initiated only upon receiving the daylight and presence value pair that is automatically sent as explained above. So in this special case we are allowed to initiate a battery requests since the device is able to process the query at this short moment. It would't work though when performing requests in an ad-hoc fashion. So we might need to consider to send this command right after when receiving any device information. The same applies for the hold-time that can be configured that basically serves as filter. More Information about the underlying protocol specification ([AG05]).

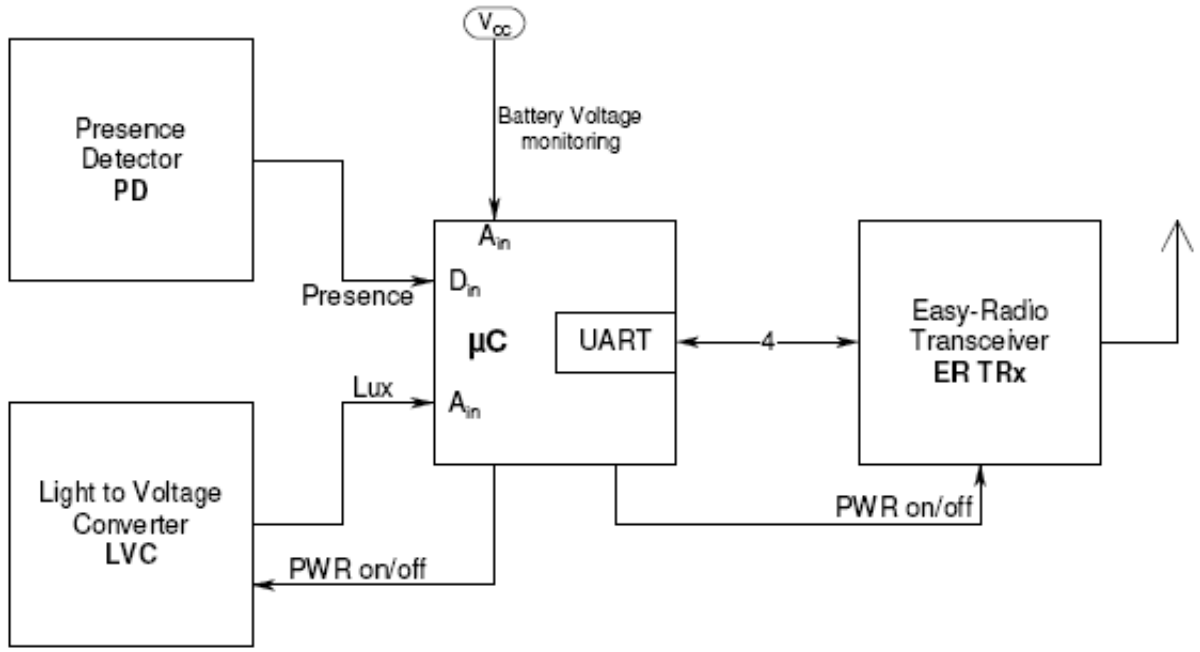


Figure 3.2: The Falcon Presence-Daylight Device

3.2.3 The Falcon Light-Actuator

The actuator is capable of switching any hooked-up electrical device i.e. A light upon receiving a specified command that has been defined by the underlying protocol specification ([AG05]).

In contrast to the Falcon Presence-Daylight Device (See section: 3.2.2) this device does provide a command set that allows the actuator to change its state. Although you cannot query the state of the actuator directly since there is no such command that would support this. On the other hand as soon as the actuator state has been affected you will be receiving a feedback message about the changed state.

3.3 Remote Control (RC)

Next to the devices and the *Falcon Communicator* the falcon system also provides a *Remote Control (RC)* (See figure: 3.3) that can be used to manually switch on and off the actuator (In example a hocked-up light). The remote control is just an additional feature that can be used of a regular wall light switch instead. Whenever the RC is being used the falcon communicator will be notified about it since the RC and the actuator are capable to directly communicate with each other without having to involve the falcon communicator in the first place. Hence the falcon communicator will let us know about this incident and

simply forward the message as well. You can convince yourself by by using one of the tools mentioned above (See: [FAL])

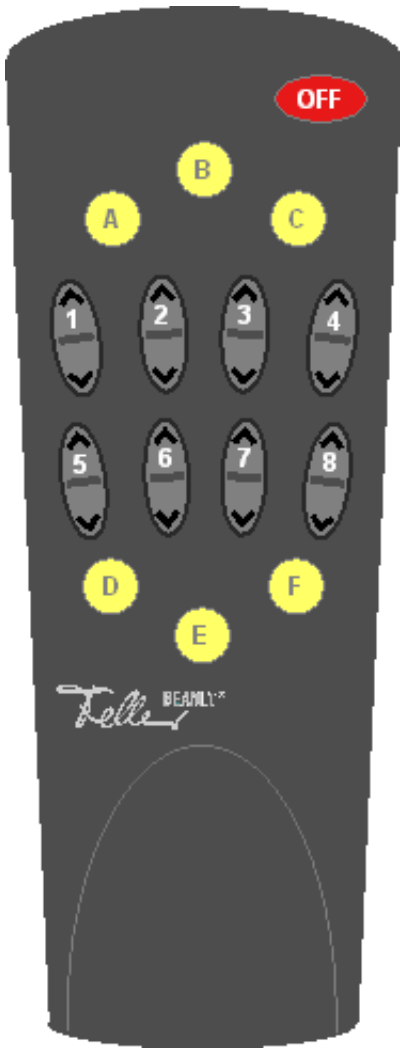


Figure 3.3: Remote Control (RC)

3.4 Property Concept

In order to counteract the issues (*Dependencies* and *Maintenance*) that have already been identified in section 2.2 we designed something more generic and practical. Something that facilitates the entire ABI System and even impacts any distributed client applications (See section: 3.5 and section: 3.9) positively. For this purpose we developed a small domain model that reflects the environment in which any further development activities may take advantage of.

Every device (physical or virtual) has a set of properties. A **Property** is actually just a tag value pair similar to that one provided by the Java API. Additionally though we supply other attributes to the property. Namely, we consider a property as readable and writeable. This is necessary since we don't know if additional devices or buses are added in the near future. And in order to provide an independent representation format such a concept was a necessary. Imagine a client software would need to know each specific supported command in advance. This would be nuts since nobody would like to rewrite any client application (See section: 3.5 and section: 3.9) or other even bundles which need to know about each specific feature of a

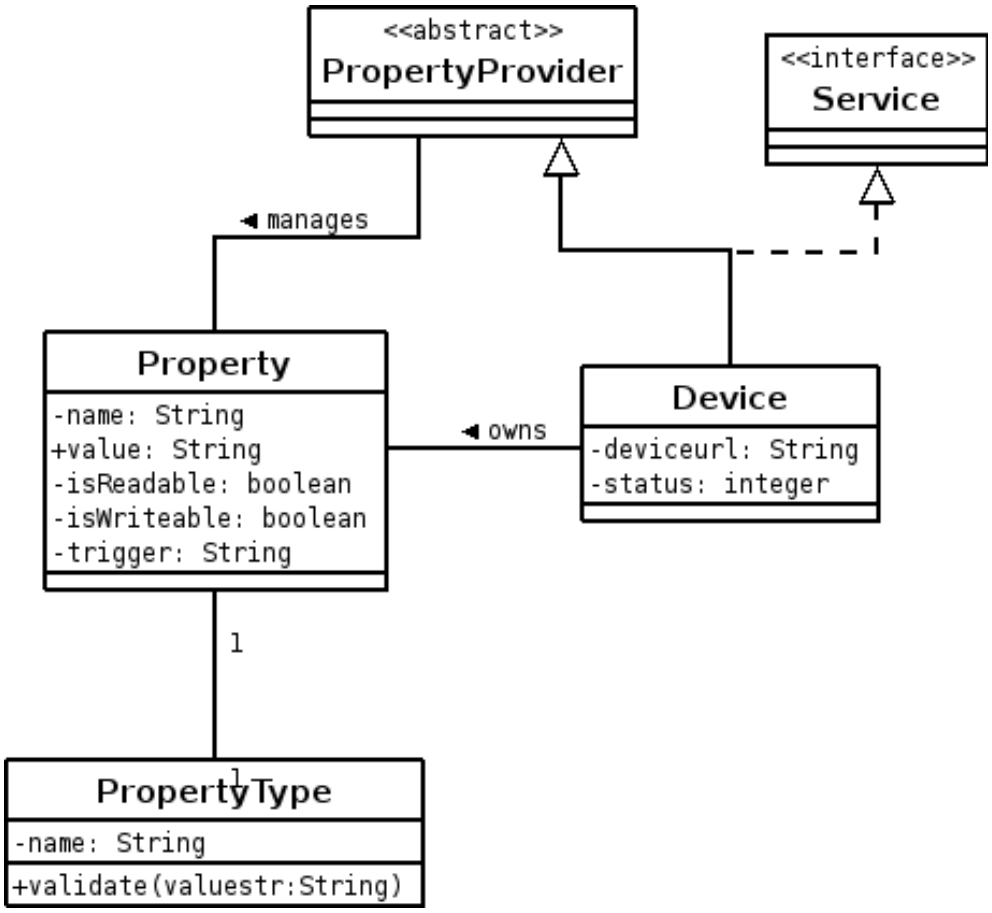


Figure 3.4: Property Concept - A Domain Model

device that might be plugged at runtime.

The benefit of this approach is that the *properties*, the *property provider* and the *property type* can now be defined in the ABI Core ([BG04a]) and are hereby detached from any concrete implementation.

So if any concrete device supports additional features it can simply add its features to its property provider since it is inherited from the property provider that manages the added properties and hereby allows a generic list of properties to be exchanged to any party. i.e. We can exchange such kind of properties in a list and send them to remotely connected clients or even across wires provided by the `Wireadmin` ([OSG]). Consumers as well as Producers are then capable to simply read out the property-list and see what kind of properties any concrete device may support.

In order to completely define a property we associated each property with a unique **PropertyType**. A **PropertyType** can hereby be represented by a set of concretely supported data types such as *IntegerType, DoubleType, EnumType*, etc.
This was crucial since a plain property value would be useless when not having the opportunity to acquire more accurate information about a specific **Property**. Quite often a **Property** value even needs to be within a specified range to be valid. Furthermore an intelligent agent actually even requires such kind of information in order to be capable to scale that value to some useful double value since not every device is providing the same scope of property value.
For instance a falcon device **Property** might provide a daylight value that needs to be a value between 0 and 127. So an appropriate min and max value might be helpful to define its validation schema or range. If we go further let's say an EnumType would need to check if the value string contains one of the values such as ON or OFF that have been defined by a device property.

Basically one can compare the entire property concept (including) the **PropertyType** with a small middleware customized for this kind of purpose. It defines a small set of an IDL in order that we know what kind of properties, each **Property** provides. For instance: A light might be able to be switched OFF and ON. Hence we provide the custom tags: writeable, readable and a set of predefined types (IDL) to define a property. With that simple locomotive a bundle or any distributed client application (See section: 3.5 and section: 3.9) can easily visualize the properties without having to know about the concrete device features in the first place. In particular GUI application can benefit from this concept since they can easily visualize the properties.
For instance:

- A light switch can be visualized, since it's writeable and can therefore be illustrated using a combobox since it has been defined as an EnumType or BooleanType.
- A presence sensor might be visualized using a label since its not writeable but readable.

Across the introduction of the property concept a new door has been opened. Namely we can now pull the strings together in the sense that a distributed client application (See section: 3.5 and section: 3.9) can now benefit from this concept also. Concretely speaking this means that an API can be developed that provides some sort of application layer that agents might be able to use instead of needing to know about the entire ABI System. So we can strike a familiar path that has already been applied in the previous distributed ABI System ([TZ03b], [TZ03a],[RS02]) again.

3.5 The Client Server Model approach

3.5.1 Overview

In order to counteract the graphical user interface issue that has already been stated in section: 2.2 we designed a client server model that considers a custom application layer protocol that defines a set of messages to be used when exchanging information between the ABI System and any custom distributed client application (agents) that need guaranteed reliable transmission of data. One major use of a custom protocol is to enable applications to retrieve changing device information and on the other hand commands which can be executed by a server similar to remote procedure calls (RPC). Hypothetically speaking a server can possibly be implemented and packed into a separate bundle within the OSGi Framework. Additionally the message protocol should further provide a standard that all distributed client applications need to adhere when communicating to the ABI System. The concept is depicted in figure: 3.5.

3.5.2 Distributed System Solutions

While there are several distributed object schemes that can be used within the pure Java environment, we'll cover a couple of solutions that we consider as a serious option for developing our distributed application model. These are:

- RMI
- SOAP
- A proprietary solution

We have following requirements that one of the solutions need to comply to.

- Client server based architecture
- The entire system should be completely network and platform independent

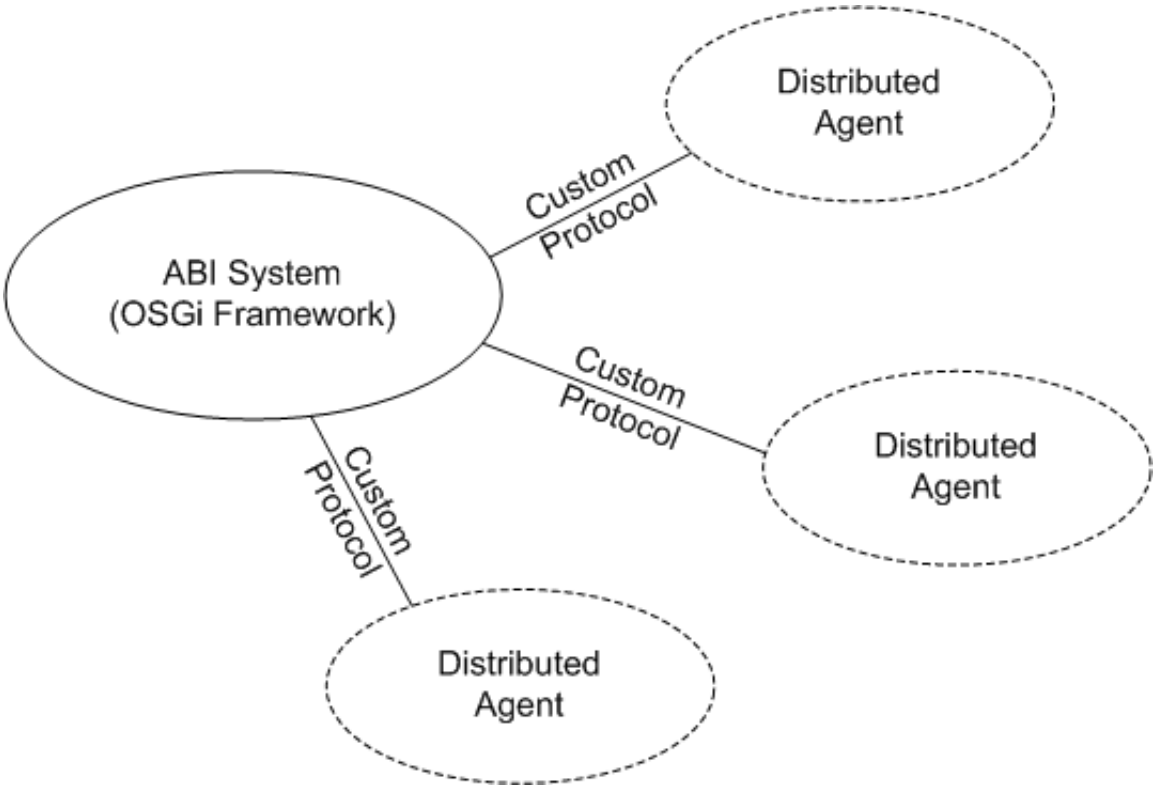


Figure 3.5: Client Server Model

- New clients should be added and removed from the system dynamically
- Clients should be able to be implemented in any programming language
- The server should be capable of sending messages to clients as well
- The applied solution should be something that can be easily installed and maintained
- Something practical and handy

All solutions have their advantages and their limitations, which we'll have a more detailed look at (See section: 3.5.2.1,3.5.2.23.5.2.3)

3.5.2.1 RMI

Remote Method Invocation (RMI) is a technology that allows the sharing of Java objects between Java Virtual Machines (JVM) across a network. An RMI application consists of a server that creates remote objects that conform to a specified interface, which are available for method invocation to client applications that obtain a remote reference to the object. RMI treats a remote object differently from a local object when the object is passed from one virtual machine to another. Rather than making a copy of the implementation object in the receiving virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the caller, the remote reference. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object. A stub for a remote object implements the same set of remote interfaces that the remote object implements. This allows a stub to be cast to any of the interfaces that the remote object implements. However, this also means that only those methods defined in a remote interface are available to be called in the receiving virtual machine [Java RMI & CORBA].

However, RMI also has the disadvantage of being relatively slow and not very fault-tolerant. Remote

methods are synchronous, which can freeze applications when the network is down. That makes RMI unsuitable for applications that need to abort communication when stalled. Although this deficiency can be worked around with the use of threads, it does not possess the flexibility of using sockets with timeouts and non-blocking I/O. Then again, non-blocking I/O is not possible with Java, so this has little bearing.

RMI is more suitable for use in the configuration part of your client application because configuration utilities are not fault-sensitive and have little need for optimized network communication. The monitoring part of your application is very likely unsuitable for use with RMI if it is a constantly running monitoring system that requires some level of real-time information delivery. Using a custom protocol over sockets will yield better results (See section: 3.5.2.3). However, if the monitoring is only single snapshot monitoring, where a user observes the status of the device only occasionally, then RMI is a good implementation approach. One other issue is that RMI is quite impractical to install and to maintain as it would be with a regular socket connection.

3.5.2.2 SOAP

The word SOAP stands for Simple Object Access Protocol. SOAP was originally developed by Microsoft, IBM, DevelopMentor and UserLand Software and was then recommended by Internet Engineering Task Force (IETF). SOAP is used to communicate between applications via HTTP using XML or extensible Markup Language. Soap is a protocol that is neither a distributed object system nor an RPC system or even a Web application, but a messaging format for machine-to-machine construction. Soap applications communicate with each other over the internet [Simple Object Access Protocol].
Some of the features worth noting are:

- Uses standard internet HTTP.
- Uses XML to send and receive messages.
- Platform independent
- Language independent
- A protocol for exchanging information in a decentralized and distributed environment

The advantages of SOAP over these protocols are that SOAP uses XML and is text based, whereas the others are dependent on object-model-specific protocols. Moreover they are not adaptable to the internet whereas SOAP uses the HTTP protocol.
In order that SOAP messages can be interpreted, a SOAP server is needed such as the Apache Axis ([Apa]). The apache Axis however is basically a SOAP engine, a framework for constructing SOAP processors such as clients, servers, gateways, etc. Therefore it requires all those components to be installed and supported. Additionally it fails in the requirement to support servers to execute remote procedure calls on the client machines. Although it would be possible since a client machine can also setup such a framework. But this would go beyond the scope of a decent installation and would be hard to get to know for third party developers as you will see later (See section: 3.9). As a short summary one can state a SOAP solution as follows:

Advantages:

- The main advantage is that it is a well-known and tested web service solution supported by i.e. the Apache foundation

Disadvantages:

1. It is not a tailored solution. As a consequence, it could not be the optimum for concrete problems
2. It needs a SOAP engine that supports an interpreter in order that SOAP messages can be recognized.
3. It needs high resources to run

3.5.2.3 A proprietary solution

A proprietary solution is clearly adaptive and can be designed to satisfy any required needs. Although following disadvantages might reflect the dark side of this solution a bit.

- A home made solution might leak of the fact of not being aboundingly tested
- Expensive in time to be developed

Since we are already familiar with developing proprietary solutions and might benefit in the sense that we might be able to apply already implemented solutions into this context we choose this as our communication technology.

3.6 Enhanced Presence Detectors

Each presence sensor that we are currently working with rather correspond to regular movement detectors than presence detecting devices since they are only capable of detecting movements of persons within a certain range. We have been realizing that if movements are very sparse or the sensors have no intervisibility with the persons itself. Thus each of the sensors were interpreting the area as unoccupied.
Of course it is possible to smooth the sensors in a certain extend but evidentially this wouldn't be very practical either since it is possible that a person is sitting at his desk while having no direct contact to the presence sensors at all.
According to the term project of Raphael Zwicker and Jonas Trindler ([TZ03a]), they were having issues with the normal presence detectors as well.
The latter solutions introduced a so called PC Presence sensor that has been developed by Tobi Delbruck (tobi@ini.phys.ethz.ch). The sensor serves as an additional presence detector that can be run as a thin client running on different workstations. The sensor might even serve as a personal presence detector since it tracks down the users mouse motions and keyboard hits.

Speaking in advise we need to consider such an option in our OSGi based ABI System as well. So the ABI System would need to consider to provide a second server that is capable of receiving such kind of information also.

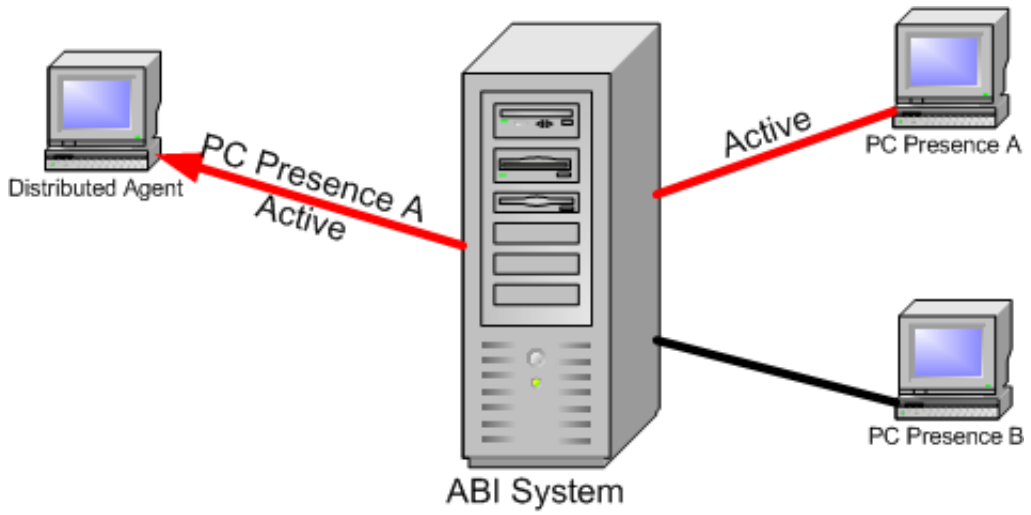


Figure 3.6: PC Presence Model

3.7 Server Discovery Service

When different servers are being plugged and are needed by the ABI System and client applications, i.e. the PC Presence server or the Remote Server it might be worth and useful to consider a discovery service that provides the capabilities of the entire ABI System in form of multicast datagrams. Hence we would achieve a less coupling between distributed client applications and the ABI System.

In order to incorporate a discovery service, we might need to consider to wrap this service into a separate bundle since it doesn't actually depend on any other services. So if any server want's to benefit from this feature it can do so by registering a `ServiceTracker` (See: [OSG]) that informs about the discovery bundle existence.

3.8 ABI Area bundle

The ABI Area refers to the issue already stated in section: 2.1.2. An area in our context should rather serve as a container that keeps count of all devices dedicated to the representing environment such as a room. It's purpose should be to provide a consistent place where such areas are being stored. Especially important when multiple clients access the different areas (See section: 3.5 and section: 3.9). Operations such as adding or removing a device and modifying other things are meant hereby. Furthermore we think that an area should be responsible of having the knowledge of each specific device location rather then the device itself (See [BG04a]). Additionally a device should not be tested if being registered since we don't want to lose any mapping relations.

3.9 General Aspects

In a first phase we want to develop a system that provides a set of services that can be used by different client applications or even bundles. In particular we want to resolve the dependencies between each bundle to be lowered and reasonably structured. Otherwise we would restrict the adaptability and the extensibility of the entire ABI System. The ABI System needs to be developed in such a way that other developers don't necessarily need to know the entire system when adding some functionality. To achieve such a system we require:

- A clear system boundary that is well documented
- Interfaces for all supported services to be well defined
- A generic principle for exchanging device information (as already mentioned in section: 3.5.1) so that underlying specific devices can be developed and are unconcerned about any other services that might consum their data.
- Services to be remotely implemented and supported by the ABI System in providing servers that allow an addressable complement. Examples of such are the PC Presence Server (See section: 3.6)
- When enabling remote services, a discovery services should centralize or "bundle" all services that the ABI System may provide.

When keeping this agreements and not allowing any violation to be taken place we should achieve a system that can be kept low coupled, stable, easy to maintain, expandable, easy to get acquainted with and most importantly we can reduce the point of failures to a minimum.

There is still one issue that needs to be addressed:
When displacing some ABI System functionality to distributed client applications we might need a standardized API that should be capable of taking down the communication part by each client application.

Furthermore it might be expanded in providing an API Specification that standardize how concrete API implementation need to be implemented in order to comply to its requirements. Most of the features might be addressed by an underlying protocol specification (See section: 3.5.1).

The purpose of such an API is to simplify any future development of custom client applications associated with the ABI System in the sense that it provides a compact application layer standard which should ultimately be implemented by a concrete API that implements the protocol requirements. This concrete implementation should provide the capability to remotely communicate to the ABI System in such a way that developers don't need to put up with the the protocol anymore. In other words it should hide any required background activities such as communicating to the ABI System (See figure: 3.7).

The benefit of such an API should be that any client application should no more depend on the complex ABI system anymore. Instead any distributed client application should be capable of controlling the ABI System in a distributed kind of fashion. In order to accomplish this we need two things to be designed and developed:

- A protocol that is responsible for exchanging ABI System relevant data such as device information, etc.
- An API standard and a corresponding reference implementation that abstracts or reproduce the entire ABI System that distributed client applications may use as their underlying application layer.

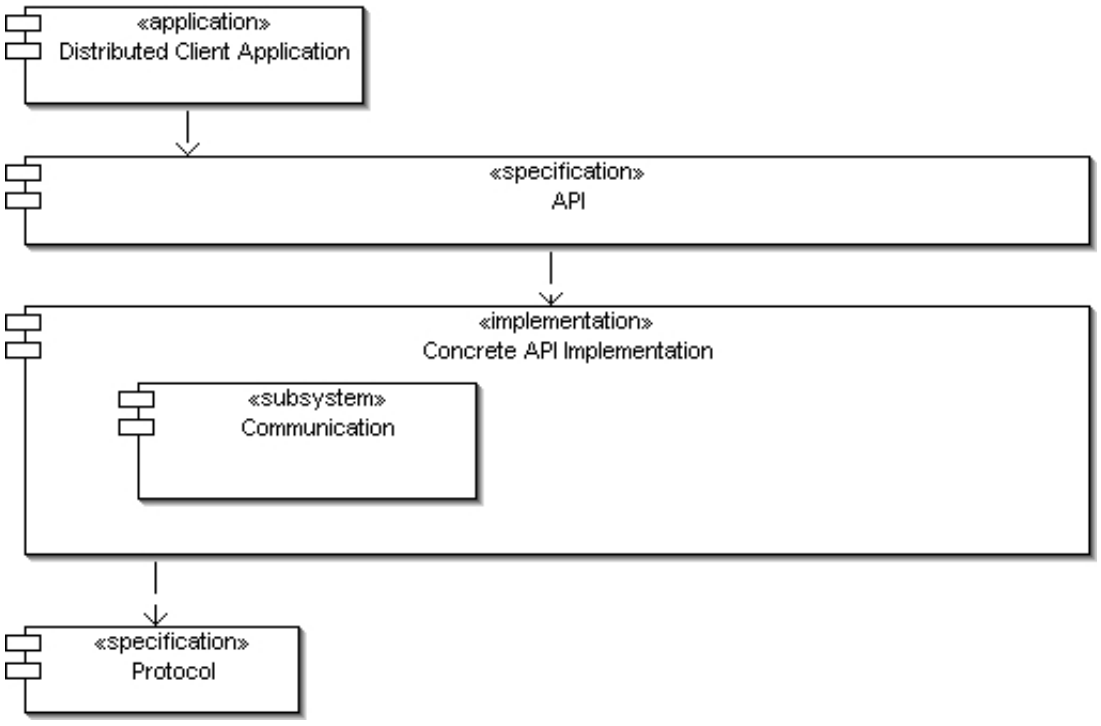


Figure 3.7: Analysis Architecture

3.10 Future considerations

3.10.1 Providing two API's

In the future we need to think of providing an implementation of the API that is capable of supporting any client applications to be installed and executed as a separate OSGi bundle as well. In other words, with an additional implementation of the API specification we should allow any client application to be either turned into a separate bundle or to be executed in a separate runtime environment, means to use a remote API implementation instead. The advantages of such an approach is to reduce the overhead that a distributed system commonly brings along.

Nevertheless for the majority a distributed client application can be a benefit in all its particulars. Especially when dealing with Graphical User Interface applications which are evidently not intended to be installed as a bundle since a window termination would cause the entire OSGi Framework to be terminated. An AI application however might benefit when being installed and utilized as a bundle rather when being run as a separate distributed client application.

For such applications in particular when incorporating intelligence control this is inevitably a better suited solution because bundles still do perform better then a client application. The benefit of this approach is that we can in one hand reduce the complexity and braking down to one single point of failure while at the same time advancing the development of applications since they can can be developed and tested completely independent and thus facilitating and satisfying both needs.

Depending on the client application such a possibility might be useful or even needed. Imagine an AI application that would suddenly stop to control an area upon a network failure. Nevertheless for the majority, a distributed client solution should fit perfectly. For an illustration see: 3.8

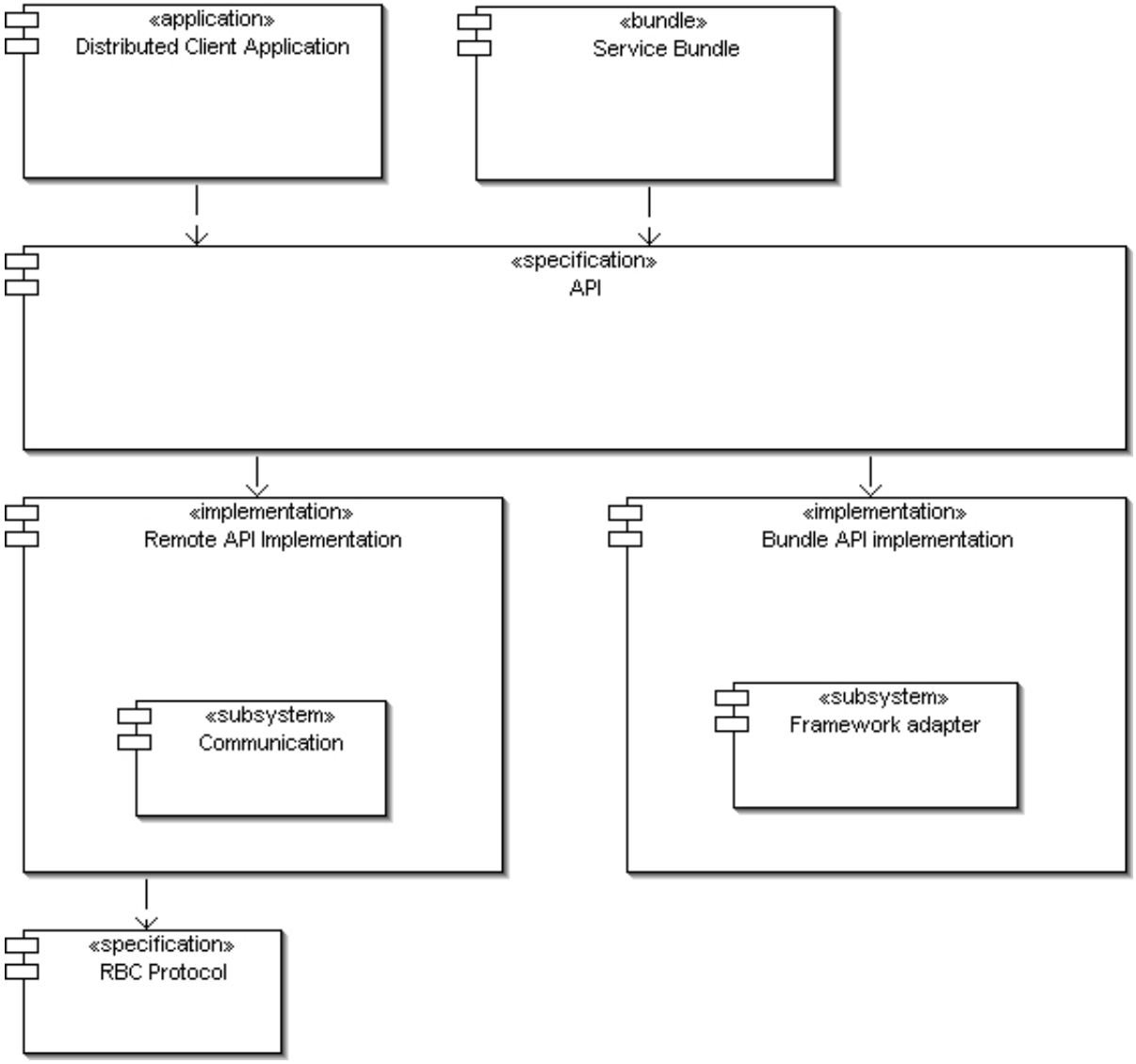


Figure 3.8: Future solution

3.10.2 LON bus integration

It should be noted that the LON technology will currently not be supported by this term project since the ABI System needs to be completely restructured and is hence no more possible to be integrated. The LON bus and all its devices are planed to be developed in the upcoming diploma thesis.
This might sound very disappointing and incomprehensible but this is really necessary since we cannot develop any possible intelligence without providing a stable ABI System in the first place. Secondly we want a System that doesn't need a couple of months to get acquainted and familiar with.

In a following diploma thesis, this term project will be expanded to a full functioning OSGi based ABI system. Which will benefit from the previously developed platform in the sense that it rather needs to concentrate in the development of different learning methodologies which are to be recorded and monitored. Additionally as mentioned the LON Bus should be integrated as well. But since the system should have been well designed and adaptive this shouldn't take that long.

Part III

Architecture

Chapter 4

ABI System Architecture

4.1 Introduction

To develop an adaptive and universal ABI System we have to define the boundary of the system. There are many different aspects and influences which characterize the architecture of such a system. This previous part point out the main influencing factors how such a system can be accomplished. The task of this chapter is to design the system as it has been suggested from the previous chapter.

We illustrated, that there are many different aims and requirements. Some of them are dependent and some independent on others. For a better understanding and to breakdown the problem, we want to divide the framework in small software units called bundles. Each bundle is responsible for one specific part of the whole system. Similar to conventional software design concepts we mostly use OSGi prescribed interfaces and develop a core system that addresses the problem of the generality so that we can reach low coupling between the bundles but altogether it is still one software.
To a certain degree we design these bundles to be developed as independent software components. In order to achieve this a bundle needs to be developed that represents the core of the ABI System that certain bundles need to adhere in order to be integrated in the ABI System. This previously developed core bundle ([BG04a]) has been more or less adapted from our predecessors. Additionally though extended with the property concept manifested in the analysis part (See section: 3.4).

4.2 System Components overview

To keep the ABI System manageable, we divide the whole system into smaller subsystems. Thereby each subsystem can be composed of a set of bundles, each of which conceive the difficulties and provide a suited solution to comply its subsystem.
We currently defined the following subsystem in our ABI System:

- *Bus and device abstraction*: Is part of the ABI Core bundle and has been partly adapted from ([BG04a])
- *Structure abstraction*: Has been implemented by the ABI Area bundle.
- *Remote control*: Has been realized by the RBC Server bundle.
- *Remote service*: Is part of the Remote Service Server bundle
- *Discovery Service*: Comprises of the ABI Discovery bundle

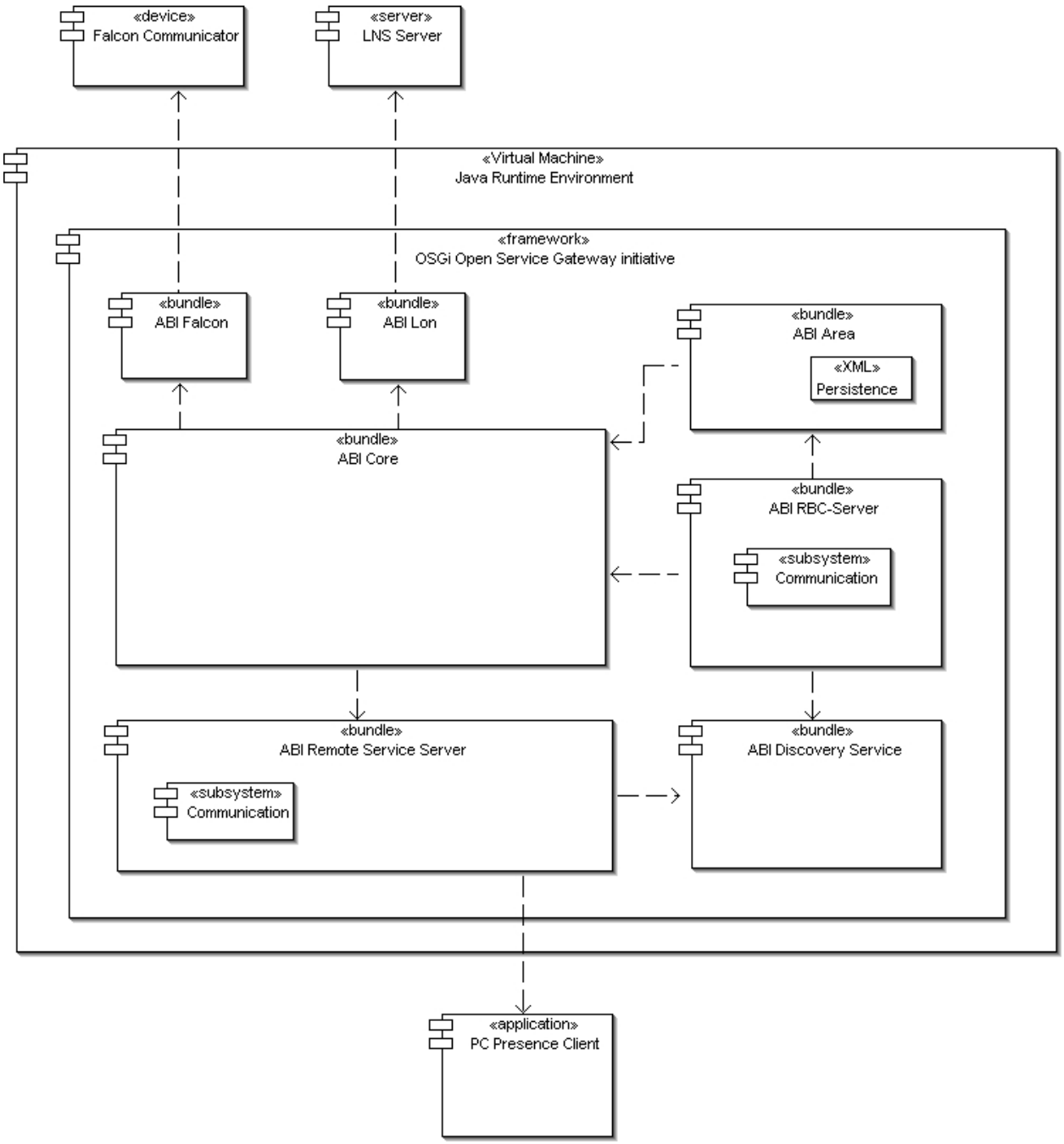


Figure 4.1: System architecture overview

4.2.1 Bus and device abstraction

The bus and the device abstraction concepts have already been implemented and applied in previous projects (See: [BG04a] and [TZ03b]). Just to refresh one's memory:
The *bus abstraction* defines a common access interface for different physical or logical bus systems. For instance for bus systems such as the LON-Bus or the Falcon-Bus.

The *device abstraction* allows a standardized access to all concrete bus devices. Each device type has its own specialized bus-specific implementation. All idiosyncrasies of a device are handled in the specific implementation of the device type.

However neither of them considered that each device may provide different features. Even though one might think that a simple presence device would only represent a device that logically provides data that implies if someone is present or not (true or false). Speaking in experience this isn't true since any device might support additional features such as a battery status or even a daylight value (See section: 3). Depending on the usage such additional features can be quite valuable to know and to provide. Imagine a battery driven presence device that would suddenly stop to respond upon being low-powered. Hence we would not be able to reliably determine the device status anymore. We can't just simple set the presence status to true since the device could have went down right after it had been indicating presence. Therefore the system would assume presence although it would't be true.
Accordingly we introduced a new concept that has already been depicted in the analysis part (See section: 3.4). Hence we don't cover this in here again. It is just important to know that the ABI Core bundle provides this feature since every concrete device of any concrete bus may make use of it (See figure: 4.2). Specifically speaking they must since services such as the RBC Server (See figure: 4.1) and others actually require this concept to be strictly adhered in order to benefit from the concept when reading out any kind of properties associated with a device when i.e. updating any distributed clients.

The boundary of the system can be defined as that each bus system should be realized by a separate bundle. i.e. The falcon bus has been realized by a bundle called ABI Falcon and basically has been implemented as a hardware specific controller. Additionally all devices that belong to a specific bus should be implemented within this bundle also. As one can see in figure: 4.2 a concrete device should implement its specific device service interface and should secondly be derived from the `PropertyProvider` since all device properties should be provided in form of properties rather then independent strings.

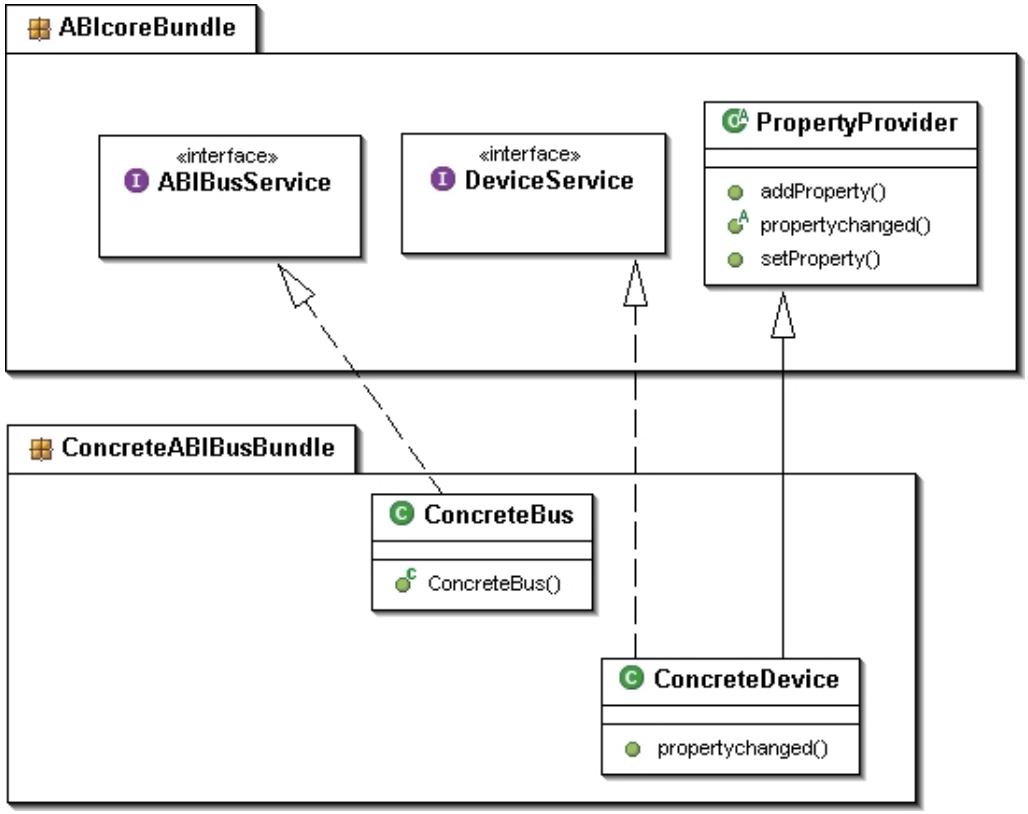


Figure 4.2: Bus and device abstraction

4.2.1.1 Property Provider

The property provider is an abstract class that all devices need to extend when willing to benefit from the property feature. The `PropertyProvider` itself has been realized with the Wireadmin concept. Hence it implements the two interfaces called: `Consumer` and `Producer` ([OSG]). It stores properties of a device and it is responsible to perform necessary updates on the wires ([OSG]): i.e. When receiving data from the devices and vise-versa to obtain update request from others(i.e. From distributed client applications), provided that the property is writeable. This class then initiates appropriate changes to the physical and virtual devices or feed the client with new data upon device property changes. i.e. Present sensor goes from false to true.

To change properties to the physical or virtual devices, this class provides an abstract method called `propertyChanged()` that all devices i.e. `FalconLightServiceImpl` need to implement.

4.2.1.2 Basic functionality

Generally a concrete device initially adds (`addProperty()`) (See figure: 4.2) its provided features in form of properties to the `PropertyProvider`.

After having registered all necessary properties the `PropertyProvider` is capable to either receive notification messages (See section: 4.2.1.1) across the wire or update all consumer wires. For instance: When a presence device receives a new presence status and will simple set the corresponding property that needs to be changed. Logically the `PropertyProvider` will update its wires and any connected complement will be receiving a common flavour called `PropertyList` (See: 4.2.1.3 that concludes of a set of properties. With this principle we can make sure that no concrete device directly exchanges any information on its own.

4.2.1.3 Modification Notification

According to section: 4.2.1.1 and 4.2.1.2 the common flavour that is being used to exchange device information is called `PropertyList`. The `PropertyList` is simply a list that contains a set of properties dedicated to a specific device (See figure: 4.3).
Depending on the device type common flavours are exchanged among bundles and later even flatten down in order to allow a text based property exchange (See chapter: 5) for distributed client applications (See chapter: 6) which reproduce the entire concept in its own independent platform (See section: 5 and 6).

4.2.1.4 Summary

In summary we can say that the bus abstraction operates as a gateway between hardware specific bus systems which might apply different technologies, such as the LON or wireless technology. In order to realize a new bus system that can be plugged to the ABI System in an ad-hoc fashion, a new concrete bundle must be implemented that must conform to the provided interfaces given in the ABI Core bundle. The concrete bundle should further implement the capabilities to communicate to the underlying hardware bus or virtual bus (See section: 4.2.4).
This is necessary because several different devices may coexist within one building. An important aspect is that each device type has its own idiosyncrasies and an its dedicated bus that addresses it. Therefore it is expedient to consider that each device should implement a property provider that standardize and defines a general concept that can later be used to exchange properties between bundles or even distributed applications.

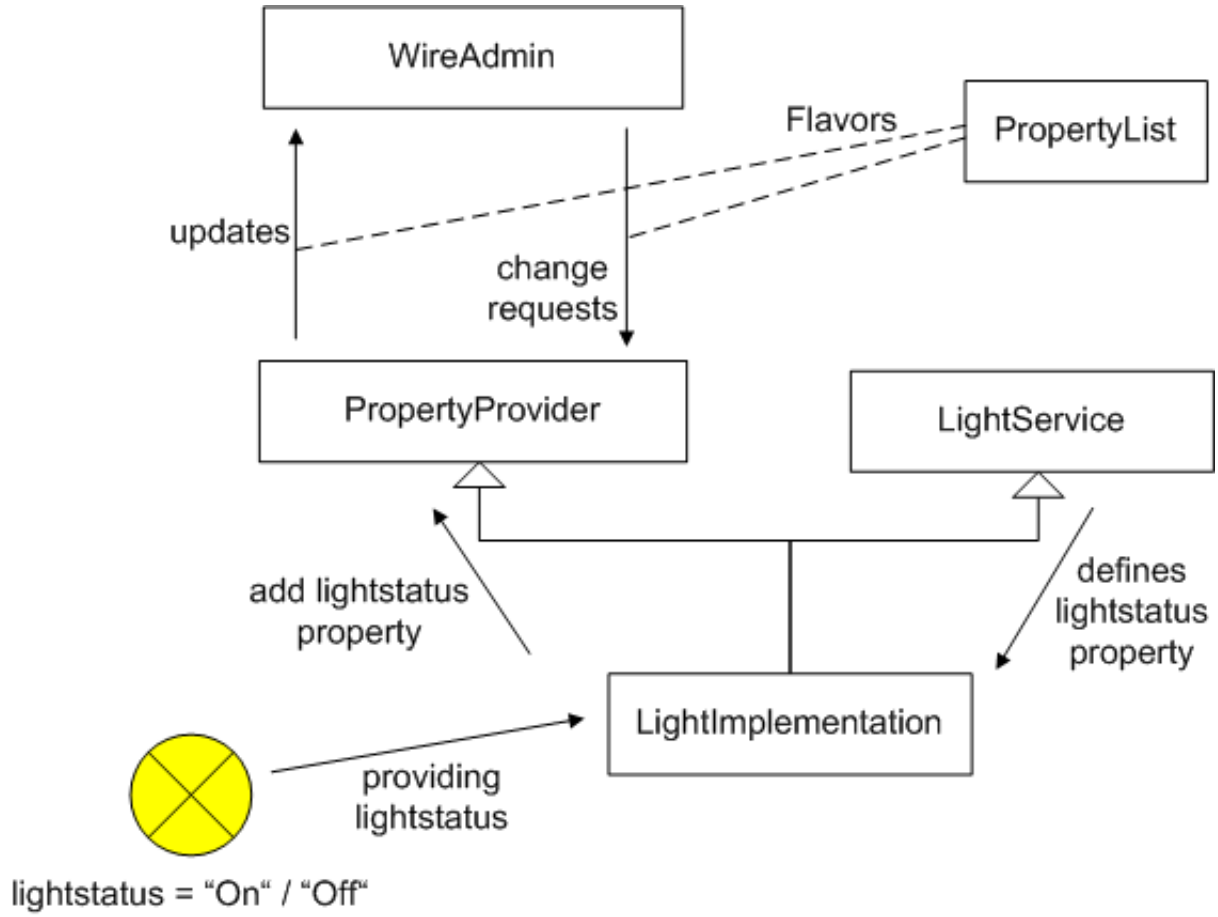


Figure 4.3: PropertyList exchange

4.2.2 Structure abstraction

The structure abstraction(area abstraction) represents a structured view to the available devices. Currently it has not been built hierarchically since we only consider separate rooms to be controlled. In the future we might consider a structure that is capable to be represented as a tree of clusters whereby each cluster can contain several devices (See: [TZ03b]). However with an appropriate naming schema one might not even require any hierarchy and can nest whenever it is required 4.4. The fact that a building structure is obviously not constant but rather dynamic in the sense that the devices within a structure may dynamically be removed, added, replaced or moved. Hence the structure abstraction must provide a further task and this is: It must administer the dynamic behavior of an environment. A further note is that there should be a way to persist the structural (area) information in some way in order to enable boot-strapping. Concretely speaking this component should never loose its content upon framework termination, etc.

The bundle that is responsible for storing and keeping track of any structural (area) information in our context is represented by the ABI Area bundle. At the first glance it might look awkward that the ABI Area bundle doesn't depend on the core and hence neither to any concrete bus bundles. This is because we don't consider devices to be registered when being added to an area.
The only collaborator of the ABI Area is the ABI RBC Server bundle (See section: 4.2.3) that actually requires the ABI Area bundle to be even launched since the server currently provides the only hatch to all provided bundle services.[1]

[1] In the future this will be extended according to section: 3.10

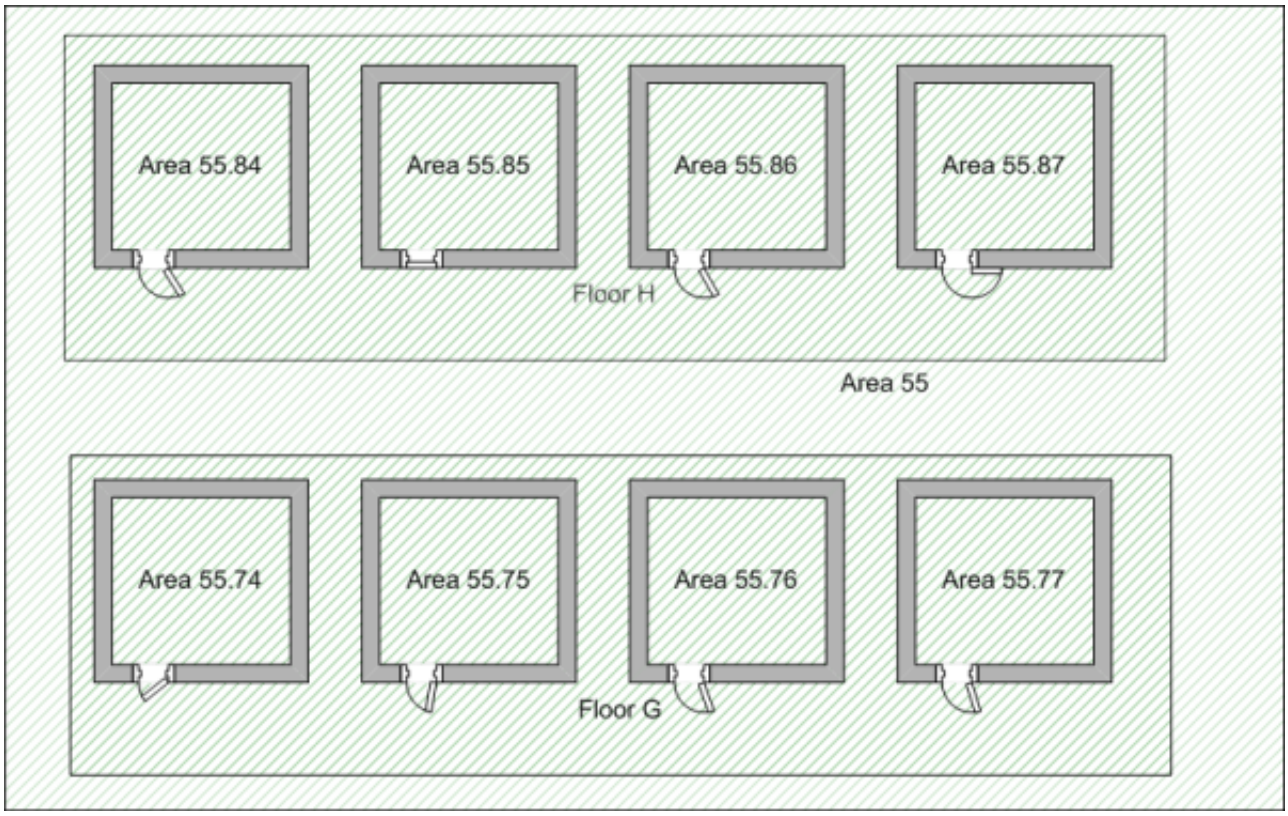


Figure 4.4: Areas hierarchy

4.2.3 Remote control

We recall that Remote control is accomplished by the ABI RBC Server bundle. Remember that we talked about a custom application layer protocol that should prescribe how messages are exchanged between the server and any distributed client application? This section discusses the server part as far it is necessary. More details about the server implementation is provided in the appropriate design section. The entire idea is depicted below in figure: 4.5. You might be unfamiliar with the acronym called RBC API ([NB05c]). So for the sake of simplicity you might want to read up about the RBC API beforehand (See chapter: 6). This might apply for the RBC Protocol ([NB05b]) as well (See chapter: 5).

4.2.3.1 Basic functionality

Basically one can say that the server is responsible for providing remote access to the ABI System. As one can see the server clearly benefits from the property concept because messages can now be defined in a generic way also. So we don't need to put up with any concrete device since we can now rely on the properties instead. It would't make sense to cover the entire server functionality in here because this would go beyond the scope of this documentation.
Broadly speaking we implemented the entire features prescribed by the RBC Protocol specification (See figure: 4.5 and chapter: 5)

4.2.3.2 Dependencies

The ABI RBC Server bundle momentarily is depending on three different bundles (See figure: 4.1). Each collaborator is described in table: 4.1.

One additional remark can be said about the ABI RBC Server. No other bundle does depend on the RBC

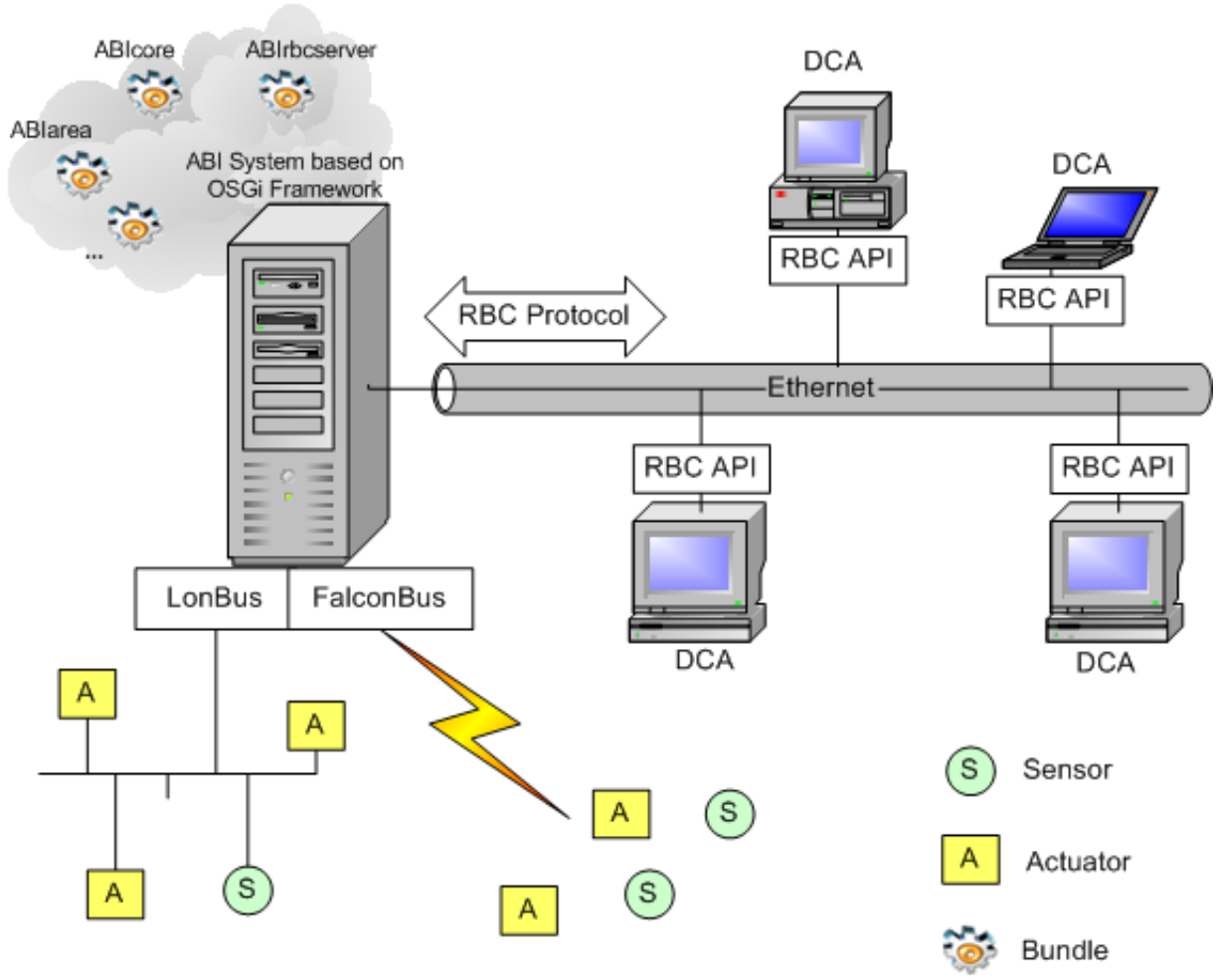


Figure 4.5: ABI RBC Server

Server since it actually doesn't provide any services to other bundles. Of course distributed client applications do depend on the ABI RBC Server in order to be capable to communicate to the ABI System.

4.2.3.3 Communication Subsystem

The communication subsystem is written to be reusable. In order to improve the re-usability, it is split into multiple components that can be exchanged, if needed.
The communication subsystem is responsible for transferring control and information messages between the client and the server.
Because the entire communication subsystem concept is quite generic it has been implemented in several variations. So other subsystem such as the ABI RBC Server and even the RBC API ([NB05c]) take advantage of this concept also. A general and overall description are given in the design part (IV)

See the protocol specification for details on the messages and their format (See chapter: 6).

4.2.3.4 Application Layer Protocol

As mentioned in the previous sections the common application layer protocol that has been used is called RBC Protocol. RBC stands for Remote Building Control and prescribes a standard that need to be adhered by both parties (Client and Server). In order to enhance the maintenance and to give the occasion to continue

Bundle	Dependency Description
ABI Area bundle	The ABI Area bundle provides a service that allows remote clients for instance to create and delete areas. Hence the server requires this bundle in order to access the service upon receiving any area messages (See chapter: 5). In order to exchange such kind of area information we provide a separate flavour called *AreaStatus*. The *AreaStatus* flavour contains a bunch of different area information. To give a practical example: When a new device has been added to an area the bundle will notify the ABI RBC Server bundle about this incident. The information can then later be broadcasted to other distributed client applications similar to *ABIDeviceStatus* (See below). More details can be obtained from the protocol specification (See: [NB05b]).
ABI Core bundle	The ABI Core bundle is needed because this bundle is the only bundle that provides some kind of virtual bus that interconnects all available buses within the ABI System. Specifically speaking it's the service called BusMultiplexingDriver that provides this capability to provide such information. Hereby we distinguish between two different type of update messages. Each of them provide two separate flavours since they make use of the Wireadmin ([OSG]). • *ABIBusStatus*: The ABIBusStatus flavour is required when sending changing bus information i.e. When a bus has been either plugged or removed. More details can be obtained from the protocol specification (See: [NB05b]). • *ABIDeviceStatus*: The ABIDeviceStatus flavour is required in particular when sending changing device information i.e. When a device has been either plugged (registered) or removed(deregistered). More details can be obtained from the protocol specification (See: [NB05b]). So this is crucial for the server to receive such kind information from the core in order to broadcast the received information to all currently connected distributed client applications (See figure: 4.5) which then have the ability to replicate a consistent small ABI System on their own (See RBC API [NB05c]).
ABI Discovery Service bundle	Among other servers this bundle provides the service that allows remote clients to discover the ABI RBC Server in the first place. In order to benefit from this service though, the ABI RBC Server must first register himself to the discovery bundle.

Table 4.1: ABI RBC Server Dependencies

this project, the documentation of the RBC Protocol ([NB05b]) as well as the RBC API ([NB05c]) have been put into separate documents rather then fitting all in one.

4.2.4 Remote service

We recall that a remote service basically refers to a service that is provided by distributed clients. In contrast to classical distributed client applications which communicate with the ABI RBC Server (See section: 4.2.3) we rather refer to clients which add a service in form of a virtual device. Currently we provide one of such. Namely the PCPresence service (See section: 3.6).

4.2.4.1 Basic functionality

The ABI Remote Service bundle basically implements two functionalities. In one hand it implements a new separate bus and on the other hand it manually builds the interface in form of a server to "remote service implementations". Because of the fact that remote services could be anything we provide a generic bus (virtual bus) for such kind of purpose that is adaptive and hence can be easily extended if more concrete remote service are hooked to the ABI System in the near future.

The concept that has been applied in this virtual bus is basically the same as the one applied in the Falcon bus. Therefore the terms: PropertyProvider, Properties and abstract bus concept should meanwhile be a familiar expression and won't need any further explanation. An exemplification of this concept is depicted in figure: 4.6.

4.2.4.2 Dependencies

The ABI Remote Service Server bundle momentarily is depending on only two different bundles (See figure: 4.1). The collaborators are quite familiar since it does have similarities to the ABI RBC Server bundle (See section: 4.2.3).
However it does not depend on the area and neither we need any kind of update message in this context since we represent a virtual device and no server that must dispatch messages in form of method calls to other subsystems. Of course we depend on the core bundle since we implement a virtual bus and accompanying virtual devices as well, but in contrast to ordinary devices we receive corresponding changes across this server and not from a bundle that replicates a hardware proxy. The second dependency that we can register is the service that allows remote clients to discover the ABI Remote Service Server in the first place. In order to benefit from this service though, the ABI Remote Service Server must first register himself to the discovery bundle.

The applied communication subsystem has been partly adapted from the communication subsystem of the ABI RBC Server since they are quite related except some few constraints and of course the fact that they don't share the same protocol (See section: 4.2.4.3).

4.2.4.3 Remote Service Protocol

The Remote Service Protocol (RSP) defines a small set of messages which are based on the RBC Protocol ([NB05b]). The differences are that it defines a complete different message set to be used.
Any other features and constraints are the same though. For instance the MessageTypes (message, request and response). So any detailed information about the entire message and its notation should be take from the RBC Protocol paper ([NB05b]).

In order not to treat this part in a separate paper we holistically define it in here (See appendix: A.1) since its similitude to the RBC Protocol.

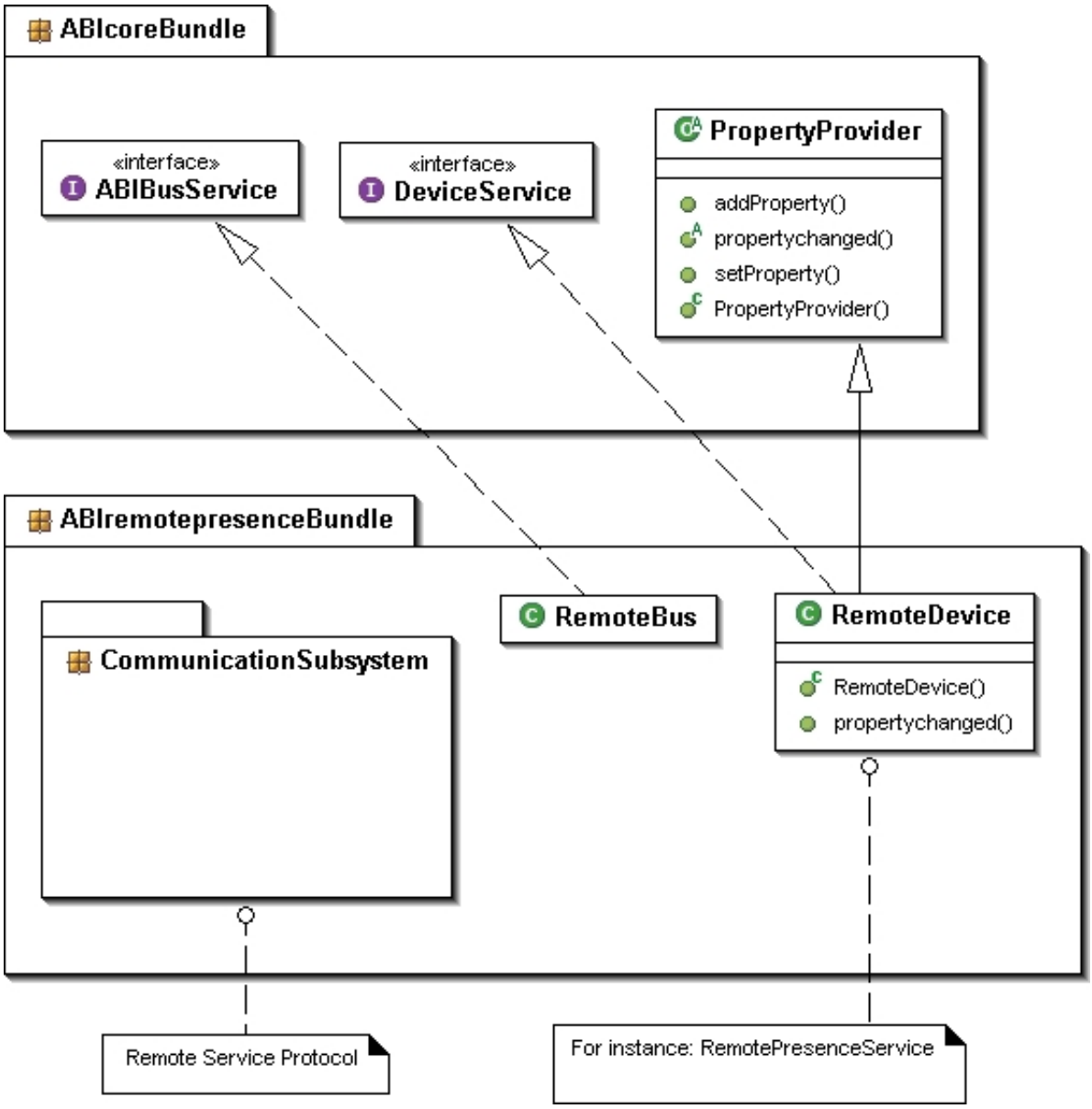


Figure 4.6: Remote Service Bundle architecture

4.2.4.4 Notes

Its important to note that the presence service is currently being tested and therefore is still experimental. However in the upcoming diploma thesis a very important feature that must properly function since we need a more "reliable" way of detecting presence according to section: 3.6.

4.2.5 Discovery system

According to figure: 4.2, the Discovery system holds a bundle called ABI Discovery bundle. The intention of the bundle has already been explained in previous sections and won't need any further clarifications. The bundle itself doesn't have any dependencies since it does not directly consume any services from other bundles.

Chapter 5

RBC Protocol Specification

As mentioned in the previous sections the documentation has been taken out of this document, in order to enhance its maintainability.
However a brief introduction to the RBC Protocol might be worth to be summarized because it might help to understand why such a protocol was necessary and in order to see how the messages have been designed.

5.1 General purpose

The purpose of the RBC Protocol is to describe the protocol to be used for exchanging information between the ABI System (RBC Server Bundle 4.2.3) and custom application programs that need guaranteed reliable transmission of data in a simple, ascii-based protocol. One major use of this protocol is to enable applications to retrieve changing device information and on the other hand commands which are executed in the RBC Server similar to remote procedure calls. Hereby it provides a standard that all ABI client applications need to adhere when communicating to the ABI System.

5.2 RBC Message Format

Having explained the basic protocol capabilities in a formal we can now define how such a packet (message) is structured in an informal way. We distinguish between three different type of messages: Request, Response and Message.
The figure 5.1 illustrates the simply structured message format that need to be adhered by both parties (Client and Server).

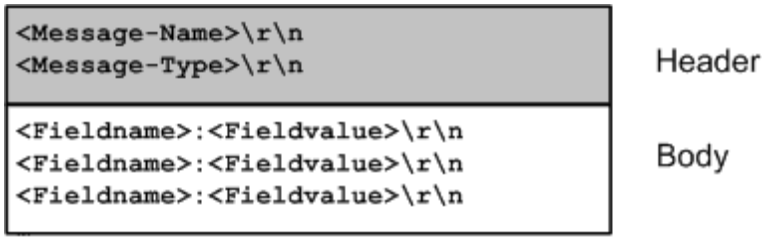


Figure 5.1: The RBC Message Format

All types of message consist of a message name and a message type, zero or more body fields, an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the body fields.
In the interest of robustness, servers SHOULD ignore any empty line(s) received where a message type or a

message name is expected. In other words, if the server is reading the protocol stream at the beginning of a message and receives a CRLF first, it should ignore the CRLF.
The CRLF at the end of the last property indicate the end of the message and MUST be adhered.

```
RBC-message = Request | Response | Message; RBC messages
```

Formally the message is defined as:

```
generic-message = messagename CRLF
                  messagetype CRLF
                  *(property CRLF)
                  CRLF
messsagename    = OCTETWCR
messagetype     = "Request" | "Response" | "Message"
```

5.3 RBC Messages

RBC messages generally consist of requests from client to server and responses from server to client. Unlike HTTP ([HTT]) this proRBC Messages a third one called message. Messages of type message can be sent from either the client or the server. The crucial difference between message to regular request-response messages is that it's implying a non-blocking one-way message.

Bundle	RBC Message category Description
Common RBC Message Packets	Common RBC Messages is a message packet category that defines special message packets which actually don't have anything to do with our context. However they provide necessary features which are quite valuable for a protocol to have when dealing with error afflicted messages and such.
RBC Control Message Packets	RBC Update Message Packets define all messages which have an associating context to control messages. Most application layer protocols support control packets that provide necessary information about clients or servers which i.e. provide remote peer information and such. So does this one. On one side ensure the compatibility and on the other to control the connection (establishment and termination). Other messages are to be included later such as the entire authentication procedure.
RBC Bus Message Packets	RBC Bus Message Packets all messages which have an associating context to buses. For instance we provide a message that requests all available buses within the ABI System (*showbuses message*) and hence also provide messages to connect respectively disconnect a given bus to / from the ABI System.
RBC Device Message Packets	Device Messages Packets define all messages which have an associating context to devices. All messages defined here are messages which are initiated by the client as either of type request-response or message.
RBC Area Message Packets	RBC Area Message Packets define all messages which have an associating context to areas. All messages defined here are messages which are initiated by the client as either of type request-response or message.
RBC Update Message Packets	RBC Update Message Packets define all messages which have an associating context to server updates. All messages defined here are messages which are initiated by the server and provided with the message type message only!

Table 5.1: RBC Messages

The RBC Protocol defines a set of different message categories, each of which has different responsibilities. These are: More details about each message category should be obtained from the RBC Protocol documentation (See: [NB05b]).

5.4 Applied example

According to the analysis part (See section: 3.4), every device does have its own set of well-defined properties. All writable properties are authorized to be set by the client if desired. In order to submit changing properties such a message or command is necessary. The RBC API ([NB05c]) uses this command when i.e. changing the lightstatus from ON to OFF or vice-versa.

In order to see how such a message look like in practice we illustrate one of them (See: 5.4).
Referring to the type constraint, this protocol defines 6 data types which can be seen as a small set of IDL. Again, each type may have associating constraints which will be appended. For more details please consider the RBC Protocol Specification ([NB05b]).

Message-Name: getproperies
Scope: Client → Server
Common Message-Type: REQ → RES
Description: The getproperies command can only be initiated the client. The command requests all available properties of a device. i.e. A light can be either switched on or switched off or Blinds can be set into different positions.

Body parameters:

1. REQ, deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...
2. RES, deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...
3. RES[1], propertyname: The custom property of a concrete device. i.e. A falcon device
4. RES[1], propertyvalue: The custom property value of a concrete device. i.e. A falcon device
5. RES[1], readable: Is the property readable?
6. RES[1], writeable: Is the property writeable? Which indicates that it can be set. i.e. A light.
7. RES[1], type: The data type of the propertyvalue. depending on the data type they may be a bunch of constraints associated with that property. For instance an integer or a float provide a {min,max} pair which define a range, etc.

Request-Response example:

```
getproperties
request
```

```
deviceurl:http://0.0.0.0:65/…
```

```
getproperties
response
```

```
deviceurl:http://0.0.0.0:65/…
propertyname:presencestatus
propertyvalue:true
readable:true
writeable:false
type:BooleanType
propertyname:daylightstatus
propertyvalue:82
readable:true
writeable:false
type:FloatType
min:0
max:127
```

[1]One single device may have several properties to be added

Chapter 6

RBC API Specification

As well as the RBC Protocol 5 the RBC API documentation has been taken out of this document, in order to enhance its maintainability.
This API will play a central role in our upcoming diploma thesis and might address you as well when you intend to develop any custom application for the ABI System.
However a brief introduction to the RBC API might be worth as well.
More details to the RBC API such as the Analysis, Architecture, Design and Implementation and the RBC Tutorial should be obtained from the RBC API documentation directly (See: [NB05c]).

6.1 General purpose

The RBC API provides an API Specification that standardize how concrete API implementation need to be implemented in order to comply to its requirements. Most of the features have already been defined by the underlying protocol specification called RBC Protocol ([NB05b]).

The purpose of the RBC API is to simplify any future development of custom client applications associated with the ABI System in the sense that it provides a compact application layer standard which should ultimately be implemented by a concrete API that implements the RBC Protocol requirements as it was prescribed by its specification. This concrete implementation should provide the capability to remotely communicate to the ABI System Server in such a way that developers don't need to put up with the RBC Protocol anymore. In other words it hides any required background activities such as communicating to the RBC Server that has been implemented and installed as a separate bundle inside the OSGi Framework([KNO]).

The benefit of such a system is that ABI client applications such as the ABI Admin ([NB05a]) can now be developed independently without having to deal with the ABI System itself. Thus improving any development practices such as complex integration testing, debugging and maintenance remarkably.
Speaking in forecast we want to develop a system that can provide the capability to plug any independent ABI client that implements different learning methodologies which can be recorded and monitored and is able to take over the control of the ABI System in a distributed kind of fashion. Thus relocating any building intelligence to client applications instead of the heavy ABI System, each of which has its own framework that is logically quite similar to that provided by the ABI System. In contrast to the ABI System though, the RBC API provides a decent level of abstraction with the result that application programmers don't even need to know about the existence of the ABI System in the first place.

Further we introduce two different implementation of the API, briefly skim how they have been implemented and most importantly provide a short tutorial that explains how they are being used in practice. Finally you will see how easy it is to write your own custom client application that can for instance be a tool for an administrative purpose such as the ABI Admin ([NB05a]) or for logging environmental data such as the the simple logging service provided by the tutorial ([NB05c]) or can even provide an entire building

intelligence software.

In the future we want to provide and implement a third implementation of the API that is capable of supporting any client applications to be installed as a separate OSGi bundle as well. In other words, with the additional implementation of the RBC API specification we allow any client application to be either turned into a separate bundle or to be executed in a separate runtime environment means to use the remote RBC API implementation instead.
The advantages of such an approach is to reduce the overhead that a distributed system commonly brings along. Depending on the client application such a possibility might be useful or even needed. Imagine an AI application that suddenly stops to control an area upon a network failure. Nevertheless for the majority the distributed system solution fits perfectly (With the usage of the remote RBC API implementation).

According to figure: 6.1, the architecture is composed into three main parts. Further the bottom is constructed out of (two) sub-parts.

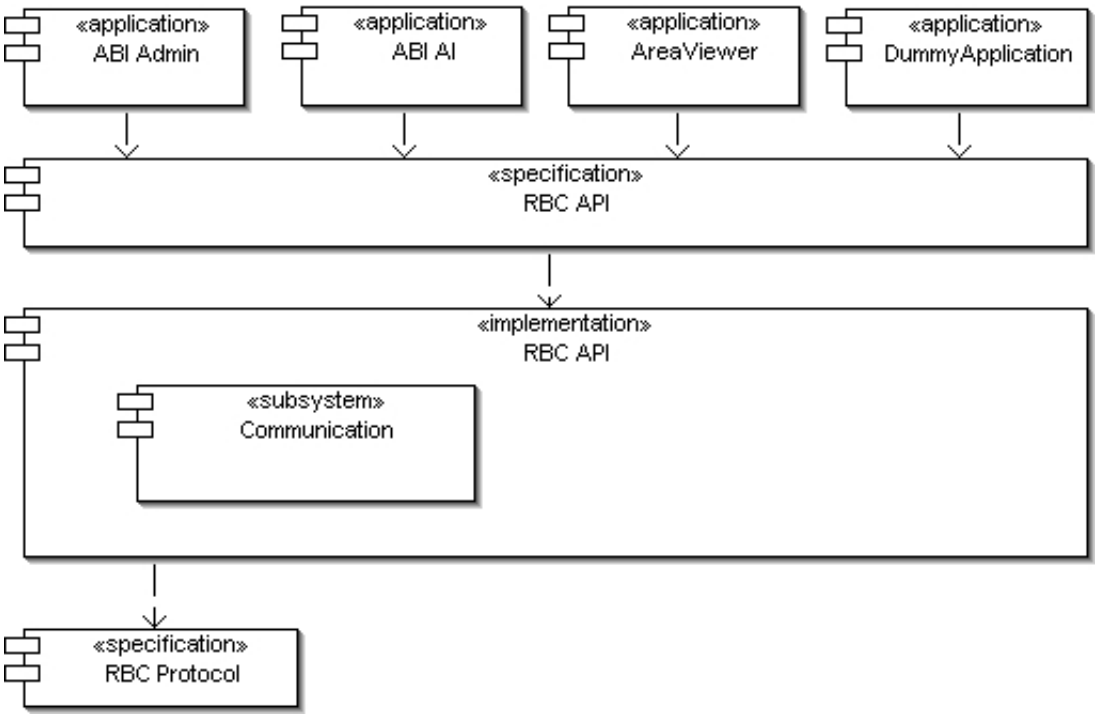


Figure 6.1: Architectural overview

Part IV

Design and Implementation

Chapter 7

Introduction

The architecture part (See part: III) rather provided the high level aspects of the ABI System, whereas the next implementation and design chapters (chapter: 8-13) are more emphasizing and revealing a more detailed view of the ABI System.

According to figure: 4.1 the entire ABI System has been constructed out of bundles. Hereby each bundle usually provides a collection of services. A service in turn is preferably an interface.

The next chapters are rather meant to be used as a reference book since they dig in quite deep into each bundle. In other words: The next chapters are intended for system developers that must keep track of the system. Each chapter has been dedicated to a separate bundle since some of them are quite big and need to be explained in how they have been designed and implemented.

Most sections within each chapter provide a mix between descriptive text, UML diagrams and code snippets. We though that it was worth to provide a combination of the three since certain things are in one way easier coded then described and on the other sometimes easier described or illustrated then provided in code snippets.

In order to provide an overview of all implemented bundles and their main responsibilities we summarized all of them (See next section: 7.1).

7.1 Bundle overview

Bundle name)	Description
ABI Core	The ABI Core can basically be considered as the heart of the ABI System. It prescribes and exposes a set of well defined interfaces that mostly correspond to either the property concept 3.4 or the abstract bus concept 4.2.1. More details should be taken from chapter: 8.
ABI Falcon	According to section: 3.2 a separate bundle must be implemented in order to embed the wireless devices into the ABI System. Hence this bundle implements a concrete bus called `FalconServiceImpl` and its supported falcon devices called `FalconLightServiceImpl` and `FalconPresenceDaylightServiceImpl`. More details should be taken from chapter: 9.
ABI Area	Section: 4.2.2 already explained how the ABI Area has been used in our context. We implemented the ABI Area to keep track of all areas that broadcast any alterations across the wires provided by the `Wireadmin` ([OSG]). Additionally we allow all areas to be stored in a XML file that can later be used to boot-strapped the entire system with its information in order not to lose any statical structure information. More details should be taken from chapter: 10.
ABI RBC Server	The fact that we allow the ABI System to be remotely controlled by any distributed client application we equip the ABI System with a corresponding server that provides some kind of entrance to the ABI System. The bundle makes use of a very adaptive generic communication subsystem that implements the RBC Protocol Specification ([NB05b]) capabilities. In particular it allows any message to be sent synchronously or asynchronously. More details should be taken from chapter: 11.
ABI Remote Service	When having a glance back to its initiator sections such as section: 3.6 you might remember that enhanced presence detectors need to be considered in the ABI System. This bundle realizes this in form of a server that in contrast to the RBC Server actively polls for its connected PC Clients ([TZ03b]) that in summary represent a small thin client software piece that can be installed on each desktop system that facilitates rather more accurate presence information then the ordinary ones. The reason why the server should poll for its clients should be taken from the corresponding chapter: 12
ABI Discovery Service	When multiple servers are being implemented and invoked in the ABI System it might be worth of considering to plug a discovery service that "bundles" all ABI System capabilities into one single multicast message that publishes *service name*, *service description*, *ip-address* and the *port* the server is running on to any possible client application. More details should be taken from the corresponding chapter: 13.

Table 7.1: ABI System Bundles

7.2 Notes

All code snippets and UML diagrams provided by the following chapters are usually incomplete and have been simplified for illustration purposes.
Additionally it should be clear that the provided code snippets are by all means not presented in their full length. Instead, all UML diagrams as well as the source code snippets are meant to provide additional information when reading the chapters.

Chapter 8

ABI Core Bundle

8.1 Overview

The design of the ABI Core bundle can basically be subdivided into three main parts. The first one is the general Abstract Bus concept which provides the generic abstract bus interface and the multiplexer that manages the device objects. The second part rather defines a set of interfaces which prescribe all currently supported services such as a presence service. The last part actually puts the generic property concept into practice that has been introduced in the previous chapters and sections.
Next to the three main responsibilities it defines some additional service such as a virtual producer that can be used as a temporary producer that send data across a temporary created wire. This can be quite useful for other bundles and services. For instance The ABI RBC Server makes use of this in order to inform certain device services about a requested status change.

The following class diagrams should depict the collaboration between the services or rather which services interfaces have been exposed for concrete services to be implement.

Please note that we currently only document the services which were necessary in reference to the Falcon devices. The generic design however will perfectly fit for the lon bus as well but since the lon part need to be incorporated into the generic design and partly rewritten, services such as the blind service must have been excluded from the current ABI System part. In the diploma thesis we will comply to that though. Please note that the sensor and actuator service interfaces are historically kept in order not to introduce something completely new. However it would also make sense to get rude of all interfaces and only make use of the `ABIBaseDevice` and the `PropertyProvider` since design technically this would be advancement. Nevertheless it might be worth to keep the interfaces for compatibility reasons and most importantly at first glance it might look weird not being capable of identifying a device by its interface. Therefore we do not add predefined properties to the `ABIBaseDevice` but rather leaf the responsibility to each separate device interface to at least allow some kind of grouping. Hypothetically speaking we may consider of introducing a concept that should determine what kind of service the entire ABI System provide. This information can then be queried by any remote client just like a webservice. If we would proceed we would end up by providing some kind of service description similar to a wsdl file that describes a webservice. The description should be human readable and should additionally contain information how to make use of the service. With this concept any service can be implemented totally independent and hence does't need to bother with any service identification issues. In summary we just need some kind of a unification that is stable. Currently we decided to take a middle course in order to be capable of heading in all direction.

8.2 Sensor services

Figure 8.2 gives an overview of the currently supported sensor interfaces. As you might have noticed, the interfaces do not illustrate much but some constants. This is because each device service currently only defines and represents a marker interface. Hereby each device service owns a set of properties which basically allows some kind of grouping among each service. According to the introduction (See: 8.1) we might need to consider something different. But momentarily we basically have no choice since we need to adapt the lon bundle in a second part. And since the lon bundle has been implemented using interfaces instead of generic properties we leave it hereby (currently).

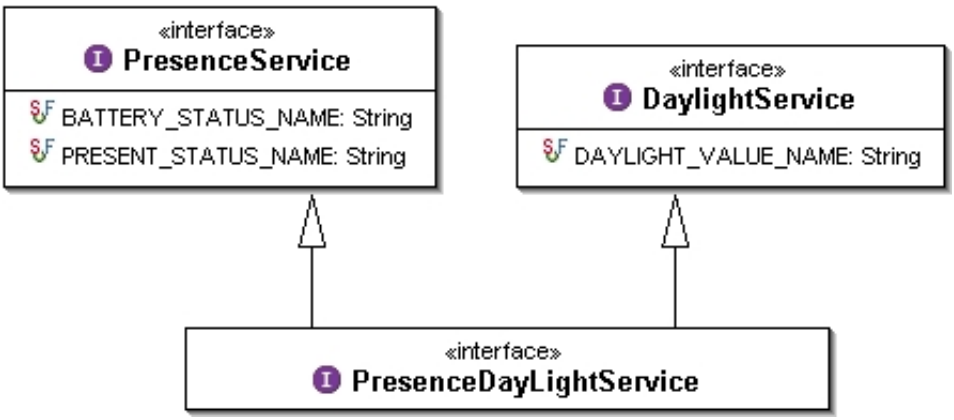


Figure 8.1: Sensor services overview

The `DaylightService` exposes an interface that each device must implement when owning a service that is capable of measuring the daylight.

```
1 public interface DaylightService
2 {
3     // The value itself should preferably be a floating point type.
4     public static final String DAYLIGHT_VALUE_NAME = "daylightvalue";
5 }
```

The `PresenceService` exposes an interface that each device must implement when owning a service that is capable of detecting presence. Please note that this could be a service that goes beyond a common presence detector. For instance a virtual device might implement some other fancy technique of sensing presence. Currently two presence services have been implemented with this interface.

- The Falcon presence daylight service (See section: 3.2.2)
- The PC Presence Service that has been realized as a Remote Service (See section: 3.7)

```
1 public interface PresenceService
2 {
3     // The value itself should preferably be a BooleanType
4     public static final String PRESENT_STATUS_NAME = "presencestatus";
5
6     // The value itself should preferably be a floating point type.
7     public static final String BATTERY_STATUS_NAME = "batterystatus";
8 }
```

The `PresenceDayLightService` exposes an interface that each device must implement when owning a service that is capable of detecting presence and measuring the daylight.

```
1 public interface PresenceDayLightService extends PresenceService,
2     DaylightService
3 {
4
5 }
```

8.3 Actuator services

As you can see we currently only support one actuator (See figure 8.2). It has been mentioned that services such as the blind service will be integrated at a later point of time since its concrete device implementation is only addressable across a corresponding lon bus implementation.

Figure 8.2: Actuator services overview

The `LightService` exposes an interface that each device must implement when owning a service that is capable of switching on and off a light.
The `LightService` prescribes a property called *lightstatus* and a corresponding `PropertyType` to be of type: `EnumyType` (See figure: 8.5.1) or consult the RBC Protocol specification ([NB05b]).

```
1 public interface LightService
2 {
3      // The value itself should preferably be an EnumType
4     public static final String LIGHT_STATUS_NAME = "lightstatus";
5
6     // The property type should therefore preferably be an EnumType
7     public static final EnumType LIGHT_STATUS_ENUMTYPE =
8         new EnumType(new String[]{"ON","OFF"});
9 }
```

8.4 Abstract Bus Concept

Each service registered as BusCategory (See figure: 8.4) gets automatically collected by the ABI bus multiplexing driver (See: [BG04a]). This automated attachment works only if the new bus service has set an appropriate unique name (BusCategory.DEVICE CATEGORY NAME) in its properties. According to the `BusCategory` interface each service that has been implemented as a bus should have its own way to connect() and disconnect() respectively.

Hence each bus that should be integrated by the ABI System needs to implement the `BusCategory` interface. Concretely speaking the Falcon Bus service and also the ABI Remote Service have implemented the `BusCategory` interface since they both represent a bus in their own ways i.e. The Falcon bus uses RS232 and the ABI Remote Service uses Ethernet. More information about the Abstract Bus Concept should be obtained from our predecessors (See: [BG04a]).

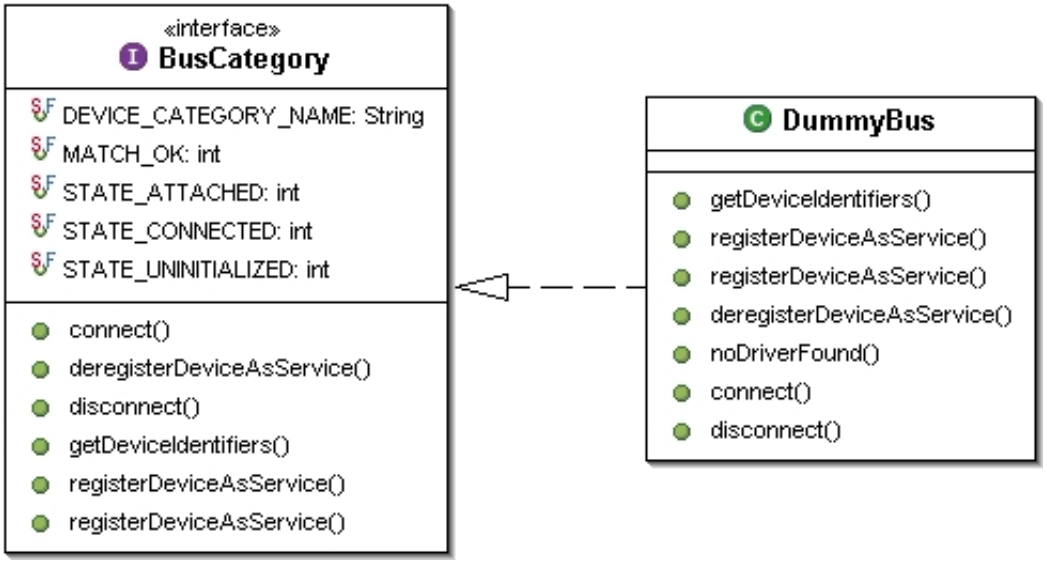


Figure 8.3: Bus service

8.5 Property Concept

8.5.1 Overview

This section provides a rather more tutorial like explanation about how properties are being used and also have been used in the current ABI System.
This section does not explain the property concept it its detail anymore. So we assume that you are already familiar with the concept in theory. In order to see how they are applied we will start with an overview that depicts all classes in order that you know how they collaborate with each other (See figure: 8.5.1).
The theory sections you might want to have a look at in parallel are: 3.4, 4.2.1.1 and 4.2.1.2. You might also want consider the RBC Protocol Specification since it makes use of the concept in the defined messages as well ([NB05b]). Currently we support following custom data types (See figure: 8.5.1).

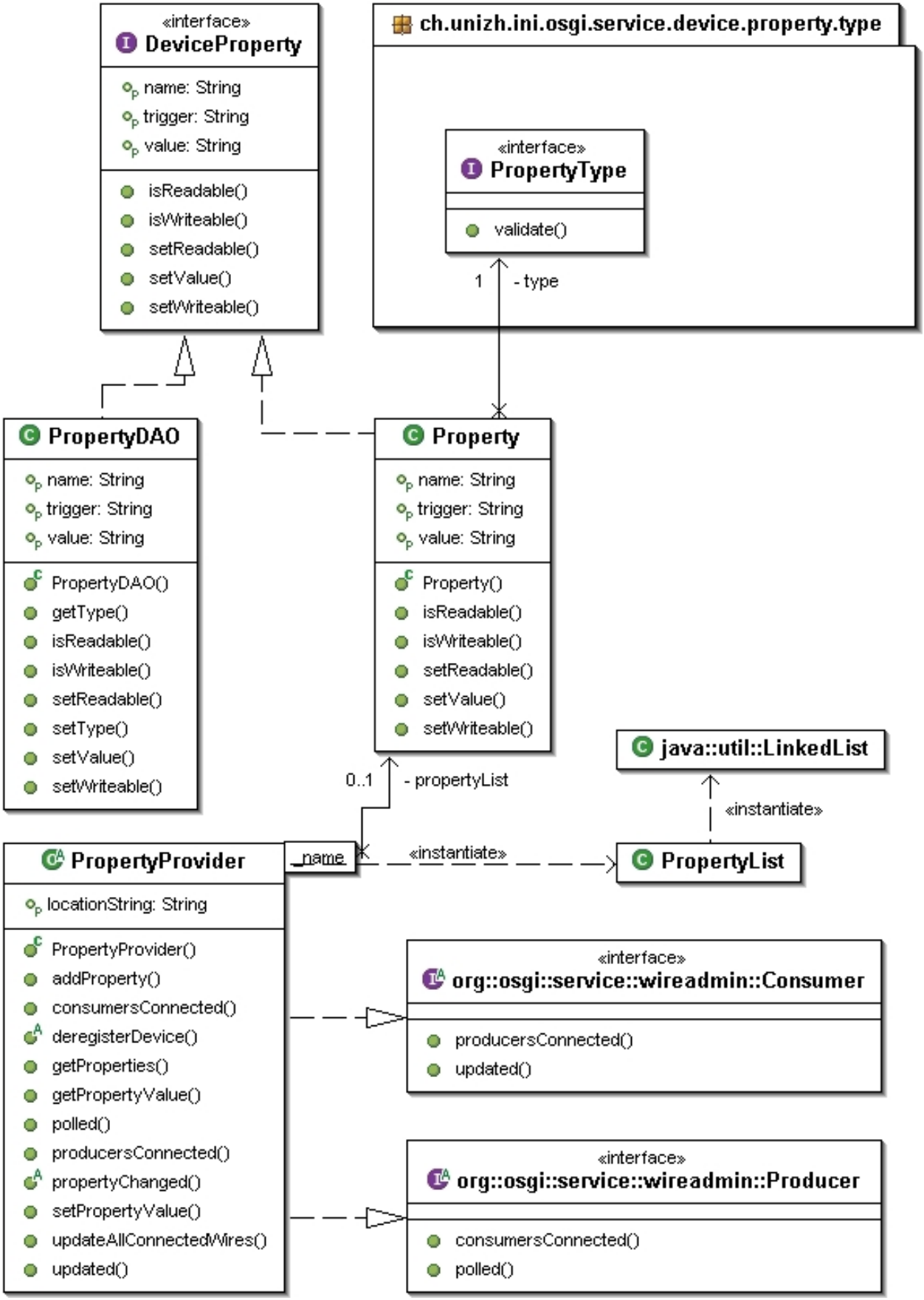


Figure 8.4: Property Concept

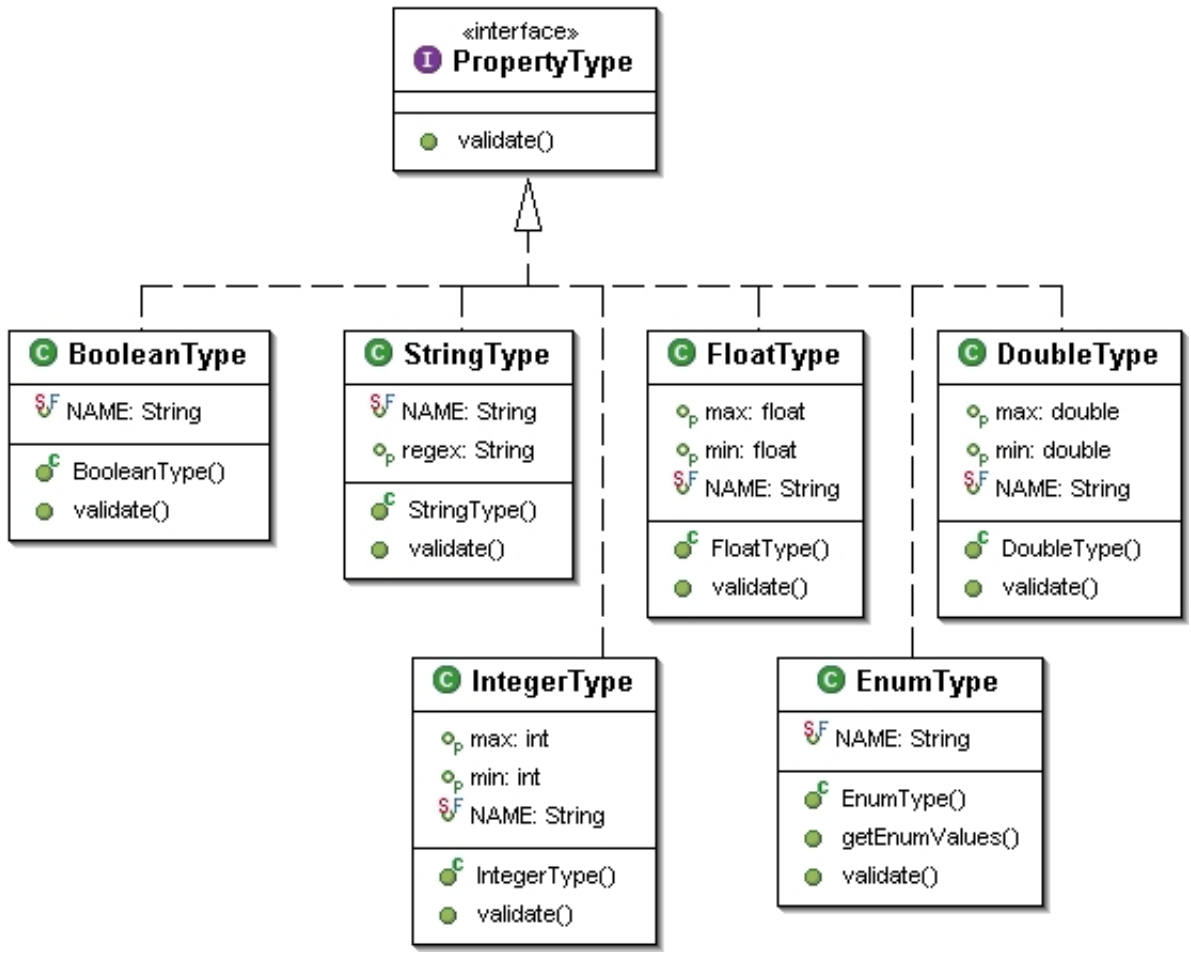


Figure 8.5: Propertytypes overview

8.5.2 Implementing a new Device

When implementing a new device you need to derive your custom device (here DummyDevice) from the `PropertyProvider` and the services you wish to support (Line 1). In this example we obviously implement a light service. So accordingly we implement the marker interface, `LightService` also. `SomeListener` refers to a fictional listener interface that provides a method called `dataReceived()`. This method will be called by the underlying hardware or virtual device upon receiving any data dedicated for this device proxy. Let us first discuss the constructor (Line 3).
As stated in previous chapter and sections each device needs to define its provided or supported properties. In this example we only provide one property. Properties are being added upon calling `addProperty()` (See below)

```
/**
 * Adds a new device property.
 * when initializing any device specific constraints. Preferably in the concrete
 * device's constructors.
 */
public void addProperty(
    String _name,         // The name of the property
    String _value,        // The value of the property
    PropertyType _type,   // The type of the property
    boolean _isReadable,  // Is this property readable?
    boolean _isWriteable); // Is this property writeable?
```

In this example we provide the propertyname LIGHT_STATUS_NAME and declare the initial propertyvalue as OFF. The third parameter has been assigned as an `EnumType` since it provides "ON" and "OFF" values. The last two parameters have been set to true since we can logically switch ON and OFF the lights (Line 9).
Herewith we have defined the device capabilities. So to recall the `dataReceived()` method from the beginning: Whenever the concrete device has sent any useful and valid data you can set the corresponding properties accordingly. This is accomplished upon calling the `setPropertyValue` method. It should be clear that the defined property status will hereby get changed.
After having set all necessary properties you can submit your changes upon calling `updateAllConnectedWires()`. As the name implies the method applies the changes through the wires provided by the `Wireadmin` (See [OSG]).
Some more implementation details about the `PropertyProvider class` can be obtained in section 8.5.3.
We are confident that you've noticed that a light can be either appear as `Consumer` or as `Producer` as well.

```
1  public class DummyDevice extends PropertyProvider implements LightService,
2          SomeListener
3  {
4      public DummyDevice(BundleContext _bc,....)
5      {
6        this.btxt = _btxt;
7        // And more initializations and registrations...
8
9        // Add custom properties that features our capabilities for this device
10       addProperty(LIGHT_STATUS_NAME, "OFF", LIGHT_STATUS_ENUMTYPE, true, true);
11     }
12
13     //...
14
15     public void propertyChanged(String name, String value, String _trigger)
16     {
17       // Send the commands to the hardware or virtual device
18       // This could be anything. Data over RS232 or Ethernet, etc.
19     }
20
21     // Method provided by SomeListener
22     public void dataReceived(/* some data */)
23     {
24       // Receive the data
25       // Analyse the data
26       // Set the properties
27       setPropertyValue(LIGHT_STATUS_NAME, LIGHT_STATUS_ENUMTYPE
28           .getEnumValues()[1], "YOUR TRIGGER");
29
30       //...
31
32       // Update all wires in the PropertyProvider
33       updateAllConnectedWires();
34     }
35 }
```

8.5.3 PropertyProvider code inspection

One of the crucial design aspect is the `PropertyProvider`. Before we dig into some implementation details we briefly summarize its task again.

The property provider is an abstract class that all devices need to extend when willing to benefit from the property feature. It stores properties of a device and it is responsible to perform necessary updates on the wires: i.e. When receiving data from the devices and vise-versa to obtain update request from others(i.e. Client), provided that the property is writeable. This class then initiates appropriate changes to the physical and virtual devices or feed the client with new data upon device property changes. i.e. Present sensor goes from false to true.¡br¿ To change properties to the physical or virtual devices, this class provides an abstract method called propertyChanged that all devices i.e. FalconLightService need to implement. In the following

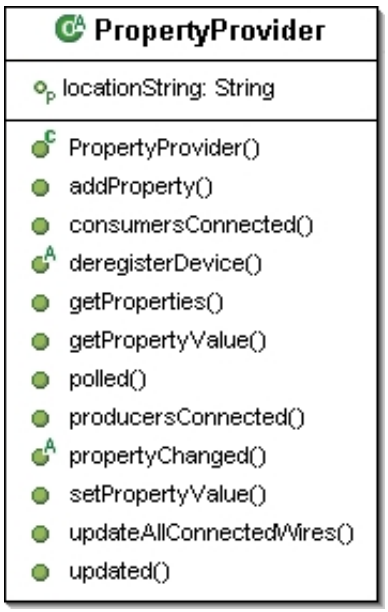


Figure 8.6: PropertyProvider

code snippets we want to illustrate how the (The `Wireadmin` and the common flavour called `PropertyList`) have been applied, we also give some insights in how the `Wireadmin` operations such as the `polled()` method has been implemented.

Since a device and hence the `PropertyProvider` can be `Consumer` as well as `Producer` we must comply both interfaces (Line 2).

```
1 public abstract class PropertyProvider implements ABIBaseDevice
2    Consumer, Producer
3 {
4 }
```

Please don't pay any attention to the `ABIBaseDevice` since it doesn't have any influence in our design but as it has been mentioned in the overview we might find something in common that each device must provide (such as a set of generic properties).

If a device consumes or is capable of producing data as well can only be determined by the corresponding device properties. Namely if a property is considered *readable*, the `Wireadmin` method `polled()` can be invoked by the consuming service or each concrete device can manually update the wires by calling `updateAllConnectedWires()`. Again, this method is called by each concrete device (provided that they own readable properties) right after they receive new device information (See section: 8.5.2)

```
1 public synchronized void updateAllConnectedWires()
2 {
3     // Create the send list
4     PropertyList sendList = new PropertyList();
```

```
5     Iterator itr = getProperties().iterator();
6     while (itr.hasNext())
7     {
8        Property p = (Property) itr.next();
9         if (p.isReadable())
10        {
11            sendList.add(p);
12        }
13    }
14    // Update the consumerWires
15    for (int i = 0; consumerWires != null && i < consumerWires.length; i++)
16    {
17        try
18        {
19            consumerWires[i].update(sendList);
20        }
21        catch (Throwable t)
22        {
23            t.printStackTrace();
24        }
25    }
26 }
```

If a property is writeable as well and hence is implying that the device can be set, i.e. a light), the `Wireadmin` method `update()` will be invoked.
Basically we would't even need to ask if we provide the flavour `PropertyList` since it can't be others then this one (Line 3). However after having received a new PropertyList across the wires (i.e. The RBC Server might have received a command the issued the request) we can read out the content and call up the `propertyChanged()` method for every property stored in the `PropertyList` (Line 18-23).

```
1  public void updated(Wire excludeSourceWire, Object in)
2  {
3     if (in instanceof PropertyList)
4     {
5        this.processProperties((PropertyList) in, excludeSourceWire);
6     }
7     else
8     {
9        //...
10    }
11 }
12
13 //...
14 private void processProperties(PropertyList in, Wire excludeSourceWire)
15 {
16    for (Iterator iter = in.iterator(); iter.hasNext();)
17    {
18        DeviceProperty p = (DeviceProperty) iter.next();
19        Property current = (Property) propertyList.get(p.getName());
20        current.setValue(p.getValue());
21
22        // Call the abstract template method of the concrete device
23        propertyChanged(current.getName(), p.getValue(), p.getTrigger());
24    }
25 }
```

8.6 Flavours

Before you read this section and its accompanying subsections we recommend to read up on the ABI RBC Server bundle first in order to understand why they are necessary (See chapter: 11).

8.6.1 ABIBusStatus flavour

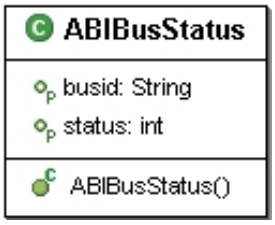


Figure 8.7: ABIBusStatus flavour

You should already be familiar why such a flavour is necessary. If not we suggest to read the appropriate architecture part (See section: 4.2.3.2).

In contrast to all other flavours this flavour does not directly define any constants in itself. This is because we adapted the previously defined class `BusCategory` ([BG04a]) which already defined the appropriate states. More information can also be found in the RBC Protocol specification ([NB05b]).

We commonly send an instance of the `ABIBusStatus` across the wire as soon as we detect when a new bus has been either plugged or removed. At the moment only the RBC Server provides a receiver class that reads out the bus status and broadcasts the the status change to all distributed clients (See section: 11.2).

8.6.2 ABIDeviceStatus flavour

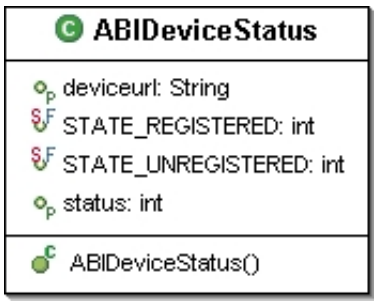


Figure 8.8: ABIDeviceStatus flavour

Similar to the `ABIBusStatus` flavour (See: 8.6.1) we assume that you are already familiar with this flavour as well. If not we suggest to read the appropriate architecture part (See section: 4.2.3.2).

This flavour defines only two constants since we only need to distinguish between STATE_REGISTERED and STATE_UNREGISTERED. Similar to `ABIBusStatus` flavour we do this upon device registration (`registerDeviceAsService`) or correspondingly when deregistering a device (`deregisterDeviceAsService`) (Both methods can be found in the `ABIBusServiceImpl` class).

Just like the `ABIBusStatus` flavour only the RBC Server provides a receiver class that reads out the device status and broadcasts the status change to all distributed clients (See section: 11.2).

8.6.3 PropertyList flavour

This PropertyList flavour is represented by the `PropertyList class`. It implements a special flavour that has already been discussed in previous sections and even chapters. So we pass on that and skip it since it doesn't provide any constants and actually can be seen as a simply `LinkedList` that has been wrapped within this class.

Chapter 9

ABI Falcon Bundle

9.1 Overview

This section discusses how we realized the controlling part of the wireless devices as well as how we finally integrated them into the ABI System.
According to the schema that has also been applied in previous projects ([BG04a]), ([TZ03b] and also has been briefly summarized in section: 8.4) a new concrete bus has been implemented that is capable of communicating to the wireless devices. Hence we implemented a specific bus that is responsible for the low level communication part that happens to be RS232 as indicated in previous sections (See section: 3.2 and section: 4.2.1). To draw a certain distinction among the different involved parts we can subdivide the ABI Falcon Bundle in two parts.

The first part is concentrating on the integration of the new concrete bus and its devices (that we call *falcon bus and falcon devices* into the ABI System (See section: 9.2) according to the exposed interfaces provided by the ABI Core (See chapter: 8).

The second part that we will be discussing in section: 9.3 is rather more emphasizing into the concrete hardware protocol specification (See: [AG05]) realization. Specifically speaking the communication part over RS232 and the threading and dispatching part.

9.2 Falcon Bus

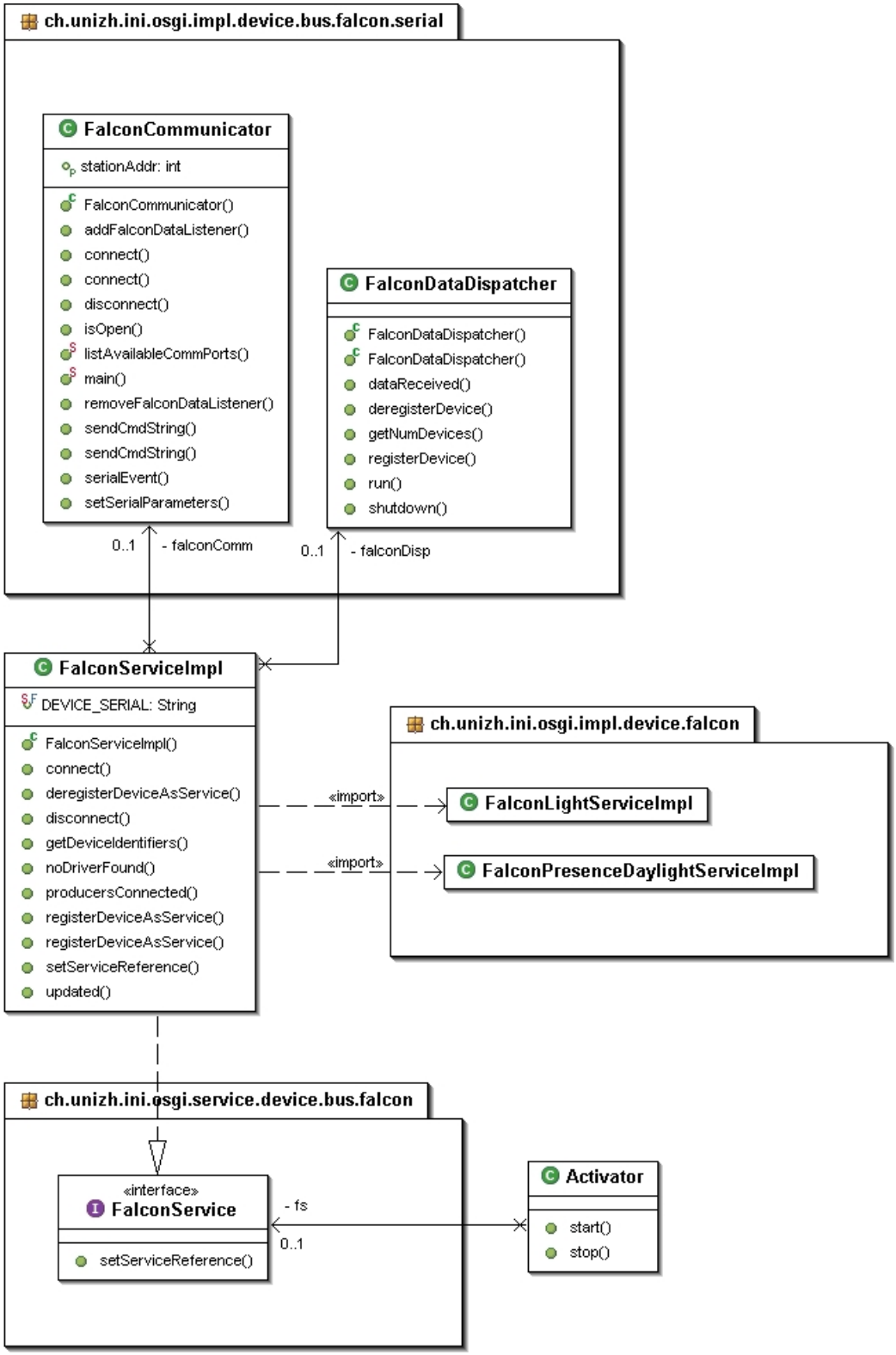


Figure 9.1: Falcon Bus

Frankly speaking the implementation schema of the bus and the devices have already been defined by the ABI Core interfaces 9.2 and thus won't need any further detailed explanation. When having a glance at

figure 9.2 you will notice a class called `FalconServiceImpl`. The `FalconServiceImpl` replicates the hardware that provides the main entry point to communicate to all interconnected devices. Hence you can think of the `FalconServiceImpl class` as a virtual proxy that has been implemented as a separate bus. When taking a close look at this class one will notice the methods `connect()` and `disconnect()`. The `connect()` method initializes and opens the comm port and prepare the devices to be ready to receive any data. Hereby we can state the bus as "CONNECTED" and proceed with any additional commands as you will see later. Inversely the `disconnect()` method closes the comm port to the falcon communicator and hereby breaks up any connections to the devices.
The following code snipped has been taken out of the `FalconServiceImpl class`:

```
1  public boolean connect()
2  {
3      //..
4      falconComm = new FalconCommunicator("COM1");
5      try
6      {
7          falconComm.connect();
8          falconDisp = new FalconDataDispatcher();
9          falconComm.addFalconDataListener(falconDisp);
10     }
11     catch (SerialConnectionException e)
12     {
13        //...
14     }
15     //...
16 }
```

Line 8 instantiates a new `FalconDataDispatcher` that allows any data to be intercepted upon its availability. We accomplish this by registering the dispatcher with the `FalconCommunicator` (Line 9).
Any further detail on the `FalconDataDispatcher` and the `FalconCommunicator` will be covered in the next section (See section: 9.3). Let's get get back to the concrete bus realization. In figure 9.2 you can depict that we support two concrete falcon devices. One device that represents a proxy for the light actuator device (See section: 3.2.3) and an other one that implements a proxy for the battery powered presence daylight device (See section: 3.2.2).

9.2.1 Falcon Presence Daylight Service

Instead of a detailed description we rather visualize some implementation aspects on how we've realized the `FalconDayLightServiceImpl class` since the `FalconLightServiceImpl class` should basically be more or less match with the example provided in section: 8.5.2.
The following code snipped has been taken out of the `FalconPresenceDaylightServiceImpl class`:

```
1  public class FalconPresenceDaylightServiceImpl extends PropertyProvider
2      implements FalconDevice, PresenceDayLightService
3  {
4      public FalconPresenceDaylightServiceImpl(BundleContext _bc,....)
5      {
6        this.btxt = _btxt;
7        // And more initializations and registrations...
8
9        // Add custom properties that features our capabilities for this device
10       FloatType floatTypeDaylight = new FloatType(0, 127);
11       FloatType floatTypeBattery = new FloatType(0.0f, 10.0f);
12
```

```
13      addProperty(DAYLIGHT_VALUE_NAME, "0", floatTypeDaylight, true, false);
14      addProperty(PRESENT_STATUS_NAME, "false", new BooleanType(), true, false);
15
16      addProperty(BATTERY_STATUS_NAME, "0.0", floatTypeBattery, true, false);
17    }
18
19    //...
20
21    public void propertyChanged(String name, String value, String _trigger)
22    {
23      // Not necessary since the presence daylight service does not have any
24      // writeable properties.
25    }
26
27    // Method provided by SomeListener Interface
28    public void dataReceived(/* some data */)
29    {
30       // Receive the data
31       // Analyse the data
32       // Set the properties
33       // Presence? Extract Presence Bit
34       if ((_data & 0x80) == 0x80)
35       {
36          // Present true
37          setPropertyValue(PRESENT_STATUS_NAME, "true", "SYSTEM");
38       }
39       else
40       {
41          // Present false
42          setPropertyValue(PRESENT_STATUS_NAME, "false", "SYSTEM");
43       }
44
45       //...
46
47       // Update all wires in the PropertyProvider
48       updateAllConnectedWires();
49    }
50  }
```

When scanning through the code snippet have a close look to line 13-16 where we set the device capabilities and also note that we don't need to implement the `propertyChanged()` method since we haven't added any properties that are writeable. Line 37 and 42 set the presence status property upon having received any valid presence status data. Of course we also need to look out for the daylight value but for simplification we skipped irrelevant code snippets. After having read and set the properties we commit the update by calling `updateAllConnectedWires()` that you should already be familiar with.
You might wonder how and where to send any data (commands) to the hardware devices. We will cover this in section: 9.3.1 since in order to be capable of sending any data to the hardware devices we need the `FalconCommunicator class`.

9.3 Hardware control

We start this section with UML class diagram that should facilitate any further readings.

The main components in the falcon communication subsystem are:

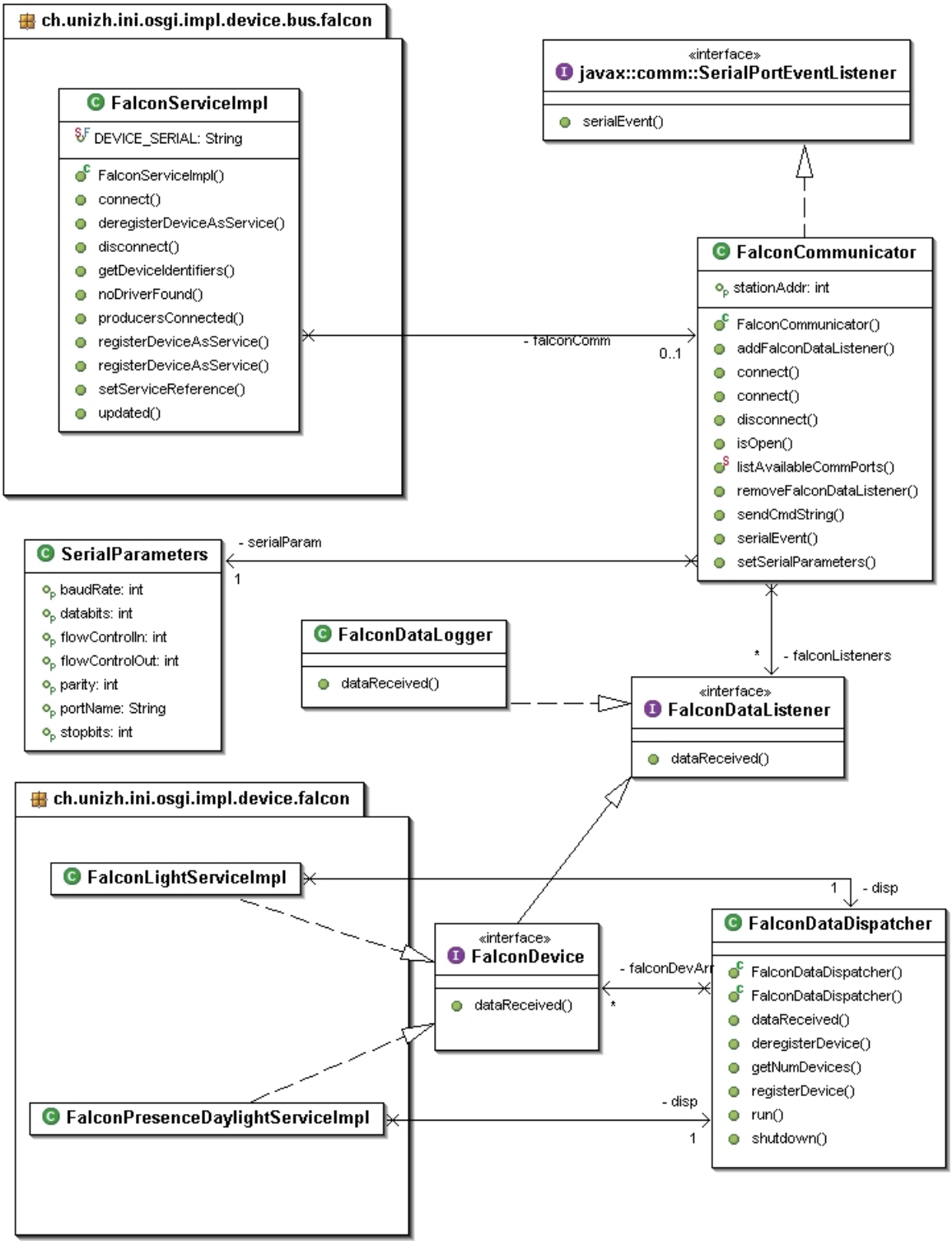


Figure 9.2: Device Communication

- FalconCommunicator (See subsection: 9.3.1)
- FalconDataListener (See subsection: 9.3.2)
- FalconDevice (See subsection: 9.3.3)
- FalconServiceImpl (See subsection: 9.3.4)
- FalconDataDispatcher (See subsection: 9.3.5)
- FalconDataLogger (See subsection: 9.3.6)

9.3.1 FalconCommunicator

The `FalconCommunicator class` is actually one of the core classes within this subsystem. Its main task is to implement the low level protocol ([AG05]) that standardize how to communicate to the devices. Because of the fact that the falcon communicator device provides an RS232 connector we accordingly implement the `SerialPortEventListener` to be capable to receive and send any command to respectively from that device. In order to establish any communication in the first place we need to adjust the comm port settings with the `SerialParameters class` to *Baudrate*: 19200, *Databits*: 8, *Stopbits*: 1, *Parity*: none.
In order to be capable of communicating over RS232 we considered the Java Communications API (See section: 9.4).

It wouldn't make sense to provide any code snippets in here since its quite complicated and would consume a bunch of slides just to explain the protocol capabilities and how we realized it. It might just worth to know that the protocol is very primitive and only supports a small set of features such as:

1. Supports a simple acknowledgment system
2. Repeat flag can be set
3. Fixed data length
4. No sequence numbering!

For more details we refer to the tools (See [FAL]) and the protocol specification ([AG05]).

The problem that one might encounter is that data that are sent over wireless might get lost. In other words: One can't really rely on the acknowledge packets since sometimes you can't distinguish which packets actually will be acknowledged. The problem we've encountered was: How do we distinguish between two messages that have been sent from the same device? At first glance this actually sounds solvable but picture this situation (See figure: 9.3.1 and figure: 9.3.1).

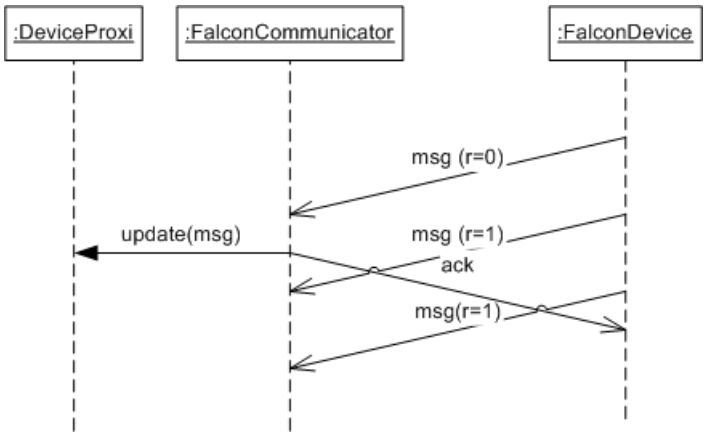


Figure 9.3: Protocol Issue SSD1

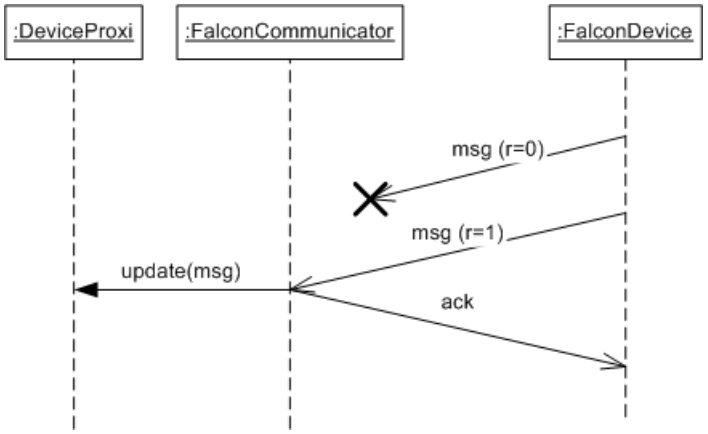


Figure 9.4: Protocol Issue SSD2

Assuming that the presence device reports presence. So we receive a packet and update the state to the concrete proxy class (in our case called `FalconPresenceDaylightService`) accordingly. After having received the new data we need to acknowledge the reception of the packet back to its source (See figure: 9.3.1). Unfortunately this might take a while and the presence device is subsequently sending the same message with the repeat flag on. This makes sense to indicate that we've obviously might have missed the first message that we actually haven't. Anyway the fact is that we should't notify the domain (`FalconPresenceDaylightServiceImpl`) about this incident anymore.
Sound logic and natural but what about the second case (See figure: 9.3.1)? What when we even miss the first announced message? We can't just drop the repeated message because we need to notify the domain since the first message has been lost. Evidentially nothing is wrong with the second case either.
The question is what if we combine both cases? Can we distinguish which of the two cases might apply?

Hence the issue we found ourselves in was how do we figure out which of the two cases apply when the second acknowledge message was destined for the first message?
The popper solution to this issue would have been to use an alternating bit protocol ([ALT]) that supports some sort of sequence numbers in the sense that it provides an alternating bit that can be set. We tried to circumvent and reduced the mis recognition by measuring the excess time of each repeated packet. If the excess time reaches a preset value we consider a message as new and fresh and hence let the appropriate proxy know about it. Luckily this case is quite rare and can practically be considered as unattainable. Nevertheless it could happen and hence need to be at least identified.

To get back to the statement that has been made in section: 9.2.1. How and where do we send any data to the hardware devices?

How When having a look at this class you will notice that we provide a method called: `sendCmdString()` that initiates a command to the desired device (provided that it can be set (writeable)). The method signature should clarify how this method should be used.

```
public void sendCmdString(int _dest, int _cmd, int _data, boolean _ack)
```

Where Each concrete falcon device proxy (`FalconPresenceDaylightServiceImpl` and `FalconLightServiceImpl`) might make use of this method. The following code snippet taken from the `FalconLightServiceImpl class` should prove that:

```
1  private boolean turnOn()
2  {
3      try
4      {
5          // Send command to the communicators
6          this.falconComm.sendCmdString(id, LIGHT_ON_CMD, true);
7      }
8      catch (SerialConnectionException e)
9      {
10         return false;
11     }
12     return true;
13 }
```

9.3.2 FalconDataListener

The `FalconDataListener` interface provides a listener interface that implemented by its collaborators such as the `FalconDataDispatcher` (See section: 9.3.5) and the `FalconDataLogger` (See section: 9.3.6). Both classes must hence be register to the `FalconCommunicator` in order to receive any incoming data packets.

9.3.3 FalconDevice

The `FalconDevice` device has actually been considered to be implemented since we wanted to distinguish between devices and non-devices (such as a logger). Currently though we only considered devices and hence only perform delegation rather them concretely implement this interface. This interface has been implemented by all concrete device proxies such as the `FalconPresenceDaylightServiceImpl class` and the `FalconLightServiceImpl class` since each of them need to be notified about any hardware device alterations.

9.3.4 FalconServiceImpl

This class has already been documented in section: 9.2 but as completion worth to be mentioned here again. In summary it implements the concrete falcon bus that abstracts the underlying wireless topology.

9.3.5 FalconDataDispatcher

The `FalconDataDispatcher` is responsible for the successful message distribution according to its source address. According to figure: 9.3 you can depict that the `FalconDataDispatcher` implements the `FalconDataListener` interface and therefore needs to implement the `dataReceived()` method accordingly.
Furthermore the entire source distribution need to be sent by a separate thread since we can't afford the RS232 thread provided by the Java Communications API to process the message in its entirety. So we perform a thread exchange to gain better performance at the RS232 connector interface. The following code

snippet illustrate how this is achieved:

```
1  public void dataReceived(int _from, int _cmd, int _data)
2  {
3      if (this.falconDevArr[_from] != null)
4      {
5          // Create packet
6          Datapacket pack = new Datapacket(_from, _cmd, _data);
7
8          synchronized (queue)
9          {
10             // Enqueue
11            queue.add(pack);
12            queue.notify();
13         }
14     }
15 }
16
17 public void run()
18 {
19    //...
20    while (queue.isEmpty())
21    {
22       try
23       {
24          queue.wait();
25       }
26       catch (InterruptedException e)
27       {
28          //...
29       }
30    }
31    //...
32    packet = (Datapacket) queue.removeFirst();
33
34    this.falconDevArr[packet.getFrom()].dataReceived(packet.getFrom(),
35      packet.getCmd(), packet.getData());
36    //...
37 }
```

9.3.6 FalconDataLogger

Has been used for testing purpose. One might consider this class when monitoring falcon message packets.

9.4 Libraries

We considered the Java Communications API to communicate to the falcon communicator because it has been tested and implemented on all major platforms such as Windows, Linux and MacOS and hence sounded very practical.
According to the Java Communications API provided by Sun Microsystem ([JAVa]) the API contains support for RS232 serial ports and IEEE 1284 parallel ports. With updated functionality, developers can:

- Enumerate ports available on the system

- Open and claim ownership of ports
- Resolve port ownership contention between multiple applications
- Perform asynchronous and synchronous I/O on ports
- Receive Beans-style events describing communication port state changes

Chapter 10

ABI Area Bundle

10.1 Overview

According to the appropriate architecture (See section: 4.2.2) and analysis (See section: 3.8) part the entire ABI Area bundle was designed to provide two public service interfaces. The `AreaServiceManager` interface that is responsible for creating and deleting areas and the `AreaService` interface that is responsible for managing an area in all its possible constraint such as adding, removing devices and even changing device locations.
The following UML diagram should depict the dependencies among the classes that make up the ABI Area bundle (See figure: 10.1). Two remarks might seem appropriate to fully understand how the ABI Area has been implemented. The first question that might be of interest is how and when do we save the areas? As you can depict from the UML diagram (See figure: 10.1) the `AreaIO` interface provides two methods called `loadArea()` and `saveArea()` which have been implemented by a concrete persistence class called `AreaIOXML`. As the name implicate this class loads respectively stores the areas when being invoked. So when the OSGi Framework initiates the bundles and within the ABI Area bundle it calls up the `start()` method that subsequently loads possibly stored areas from the file into the framework. Inversely when the framework terminates we save the current area context into a XML file. The following code snippets from the `Activator` should illustrate that.

Load the XML file upon calling `start()`:

```
1  // Load stored areas
2  AreaIO io = new AreaIOXML(bc, "testareas.xml");
3  try
4  {
5      io.loadArea();
6  }
7  catch(PersistenceIOException _exception)
8  {
9      // drop usually happens when the file does not exist.
10 }
```

Save area content to a XML file upon calling `stop()`:

```
1  // Before we stop we save all areas and its added devices
2  AreaIO io = new AreaIOXML(this.bc, "testareas.xml");
3  io.saveArea();
```

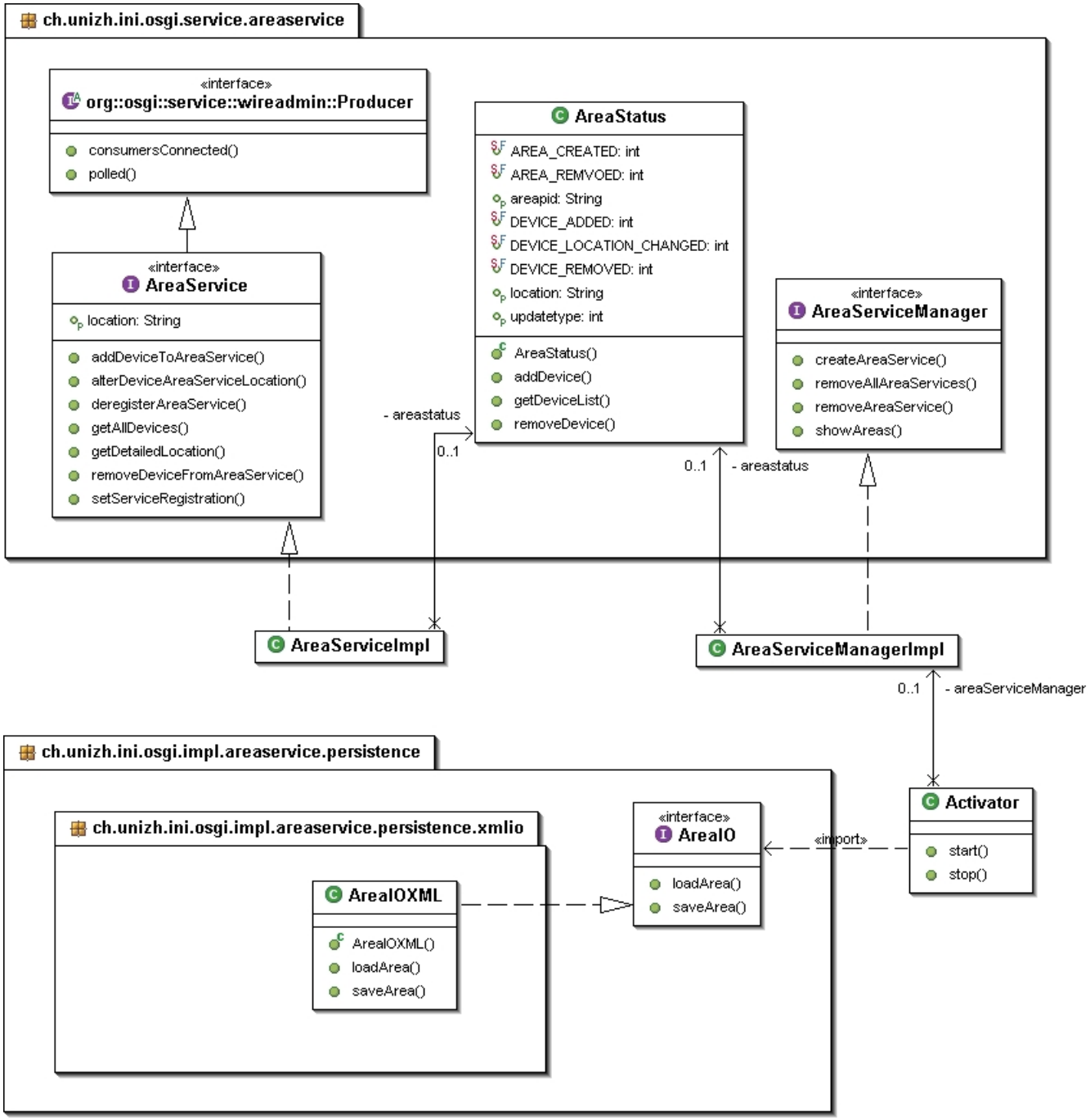


Figure 10.1: ABI Area Overview

10.2 Area updates

In this subsection we briefly outline how we notify distributed client applications about a new area creation. The method `createAreaService()` (method of class `AreaServiceManagerImpl`) is commonly executed by the dedicated message handler called `CreateAreaHandler` (See chapter: 11). When this method is once being invoked we can observe following activities.
Code snippets from the `AreaServiceManagerImpl class`'s `createAreaService()` method.

```
1  //...
2  Properties props = new Properties();
3
4  String[] clazzes = { AreaService.class.getName(),
5    Producer.class.getName() };
6
7  // Set producer flavor to AreaStatus
8  props.put(WireConstants.WIREADMIN_PRODUCER_FLAVORS,
9  new Class[] { AreaStatus.class });
10
11 // Create AreaService
12 AreaService area = new AreaServiceImpl(bc, _areaPID,_location);
13
14 // Store PID in properties
15 props.put(org.osgi.framework.Constants.SERVICE_PID, _areaPID);
16
17 // Register AreaService to the framework
18 reg = bc.registerService(clazzes, area, props);
19
20 area.setServiceRegistration(reg);
21
22 areastatus = new AreaStatus();
23 areastatus.setUpdatetype(AreaStatus.AREA_CREATED);
24 areastatus.setAreapid(_areaPID);
25 areastatus.setLocation(_location);
26
27 for (int i = 0; consumerWires != null && i < consumerWires.length; i++)
28 {
29    consumerWires[i].update(areastatus);
30 }
31    //...
```

Line 2-9 Defines the area as `Producer` since we need to produce a message with a common flavour called `AreaStatus` (See section: 10.3).

Line 12 Instantiate a new area with the provided areapid.

Line 22-25 Creates a new AreaStatus object and sets the type of update accordingly to AREA_CREATED (See section: 10.3)

Line 27-30 Updates the wires of the the `AreaServiceManagerImpl`. The complement `Consumer` service is a class called `AreaUpdateDispatchImpl` that according to section 4.2.3.2 and section 11.2 must be located in the RBC Server bundle (See chapter: 11)

10.3 AreaStatus flavour

When having a glance back to figure: 10.1 you might have noticed the class called `AreaStatus`. You should know by now what the flavour is for. If not we suggest to read the appropriate architecture part (See section:

AreaStatus name)	Description
AREA_CREATED	Needs to be set when a new area has been created. Value = 0x01
AREA_REMVOED	Needs to be set when a new area has been removed. Value = 0x02
DEVICE_ADDED	Needs to be set when a new device has been added to an area. Value = 0x03
DEVICE_REMOVED	Needs to be set when an existing device has been removed from an area. Value = 0x04
DEVICE_LOCATION_CHANGED	Needs to be set when the device location of an existing area has been altered. Value = 0x05

Table 10.1: AreaStatus flavour

4.2.3.2).
To get a visual what each of the defined constants (name and value) might impact when being set by one of the ABI Area bundle classes (in example see the code snippet on line 23 in section 10.2) we give a short explanation about them (See table: 10.1):

10.4 Area XML file

It would't make sense to dig into the code that loads and stores the areas but to get a hunch on how such a file look like here a small excerpt of it.

```
<?xml version="1.0" encoding="UTF-8"?>
<areas>
  <area pid="http://55.G.71" location="55.G.71">
    <device
        deviceurl="http://0.0.0.0:64/falcon.bus-1.0/..." detailedlocation="anywhere"/>
    <device
        deviceurl="http://0.0.0.0:35/falcon.bus-1.0/..." detailedlocation="window"/>
  </area>
  <area pid="http://55.G.74" location="55.G.74">
    <device
        deviceurl="http://0.0.0.0:32/falcon.bus-1.0/..." detailedlocation="window"/>
    <device
        deviceurl="http://1.1.1.1:4/remote.bus-1.0/..." detailedlocation="laptop"/>
    <device
        deviceurl="http://0.0.0.0:35/falcon.bus-1.0/..." detailedlocation="window"/>
    <device
        deviceurl="http://0.0.0.0:65/falcon.bus-1.0/..." detailedlocation="corridor"/>
  </area>
</areas>
```

10.5 Libraries

In order to be capable of parsing a XML file we considered the well known Xerces Parser implementation provided by Apache ([XER]).
It has been chosen because of it's superiority over the integrated parser provided by Sun Microsystem. To quote its capability.

The Xerces Java Parser 1.4.4 supports the XML 1.0 recommendation and contains advanced parser functionality, such as support for the W3C's XML Schema recommendation version 1.0, DOM Level 2 version 1.0, and SAX Version 2, in addition to supporting the industry-standard DOM Level 1 and SAX version 1 APIs ([XER]).

Chapter 11

ABI RBC Server Bundle

11.1 Overview

The RBC Server bundle is presumably one of the core subsystems in the entire ABI System. The RBC Server again basically is the main entry point for all distributed client applications in the sense that it provides a set of public methods that can be remotely executed. Therefore the name of genesis: *Remote Building Control* (RBC) Server. In the center it provides a subsystem that is responsible for receiving and sending messages to respectively from clients. Next to the communication subsystem we subdivided the ABI RBC Server bundle into two subparts.
One part that implements the piece that receives changing bus, area, device and device property information that must be broadcasted in form of update messages (as mentioned in section: 4.2.3.2) to other clients.
The other part is responsible for implementing the entire protocol specification in form of a communication subsystem that provides asynchronous message handlers each of which are responsible for handling a specific message type and hence does interact with other bundle services in order to comply its dedicated request.
Before digging into the communication subsystem we first outline how the first subpart is composed of. Since we use wires to interconnect other bundle services rather the using their interfaces directly in order to achieve low coupling, one of the collaborators that has been used is the the `Wireadmin`. When having a glance back to table 4.1 where we've outlined the dependencies among the bundles, you will notice that other bundles basically act as `Producer` whereas the server represents a `consumer`. You can observe this when studying figure: 11.1. Please note this diagram is incomplete and has been simplified since we want to get to the bottom of how they collaborate rather then all the details.

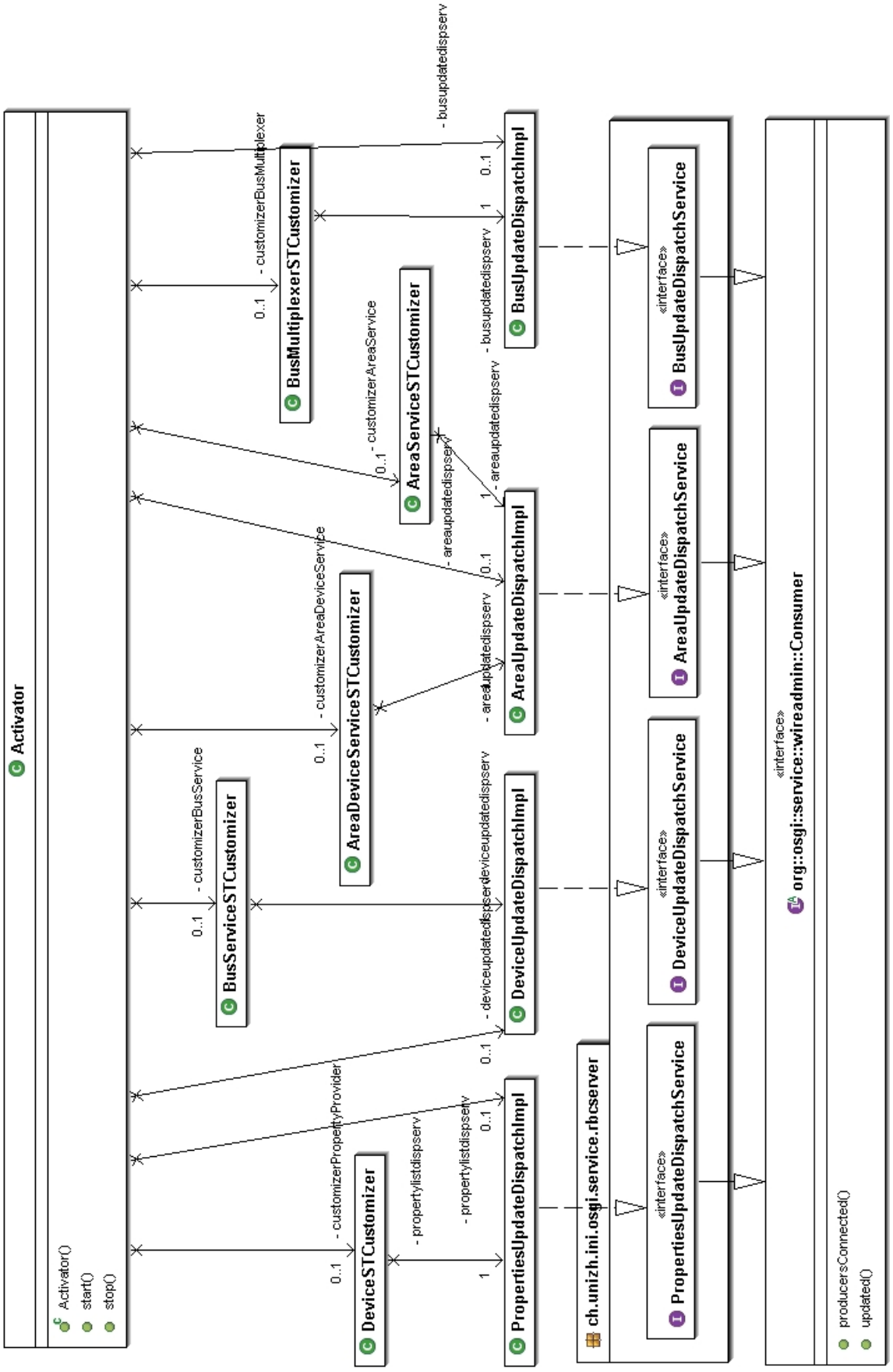


Figure 11.1: RBC Server Overview

Service Trackers (ST's)	Description
AreaDeviceServiceSTCustomizer	The `AreaDeviceServiceSTCustomizer` is a ServiceTracker that tracks devices within an area. For instance: When a new device has been added to an area
AreaServiceSTCustomizer	In contrast to the `AreaDeviceServiceSTCustomizer` this `ServiceTracker` tracks an area itself. For instance: When a new area has been created or removed.
BusServiceSTCustomizer	The `BusServiceSTCustomizer` is a `ServiceTracker` for tracking specific BusMultiplexer activities such as busupdates. For instance: A new bus has been plugged or the state of the bus has been changed through a client.
BusMultiplexerSTCustomizer	In contrast to the `BusServiceSTCustomizer` this `ServiceTracker` is responsible for tracking devices. We recall that the `BusMutliplexer` is know by the core only. In practice this means that whenever a new device has been either added (registered) or removed (deregistered) to/from the ABI System, this `ServiceTracker` will be informed.
DeviceSTCustomizer	According to figure: 11.1 this `ServiceTracker` tracks device property changes.

Table 11.1: Service Trackers

11.2 Service Trackers

According to figure: 11.1 you can depict that various focal points have emerged into some kind of schema called `ServiceTracker` that is supported by the OSGi Framework ([OSG]). According to the OSGi Specification a `ServiceTracker` can be used to track other services of their existence or non-existence.
Hence this is exactly what we've needed in our context since we use such a `ServiceTracker` for instance to get notified about any device adds or removes. Concretely speaking this means when a device has been added to the ABI System we pull a wire to it and inversely we delete a wire to a recently removed device. Of course we need to cover up more cases then just adding and removing devices. Table: 11.1 itemizes each of the implemented trackers and describe what they track and what they do.

In order to see how this schema works in practice we illustrate some code snippets of the `DeviceSTCustomizer`.

Each `ServiceTracker` such as the `DeviceSTCustomizer` have set of methods (See figure: 11.2).

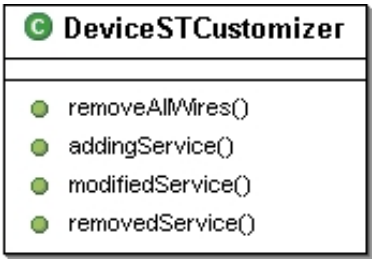


Figure 11.2: DeviceSTCustomizer

One of them is the `addingService()` method that in our usage creates a new wire as soon as we detect a new PropertyProvider and hence a new device (Line 13).

```
1  public Object addingService(ServiceReference reference)
2  {
3    PropertyProvider service = (PropertyProvider) bc.getService(reference);
```

```
4
5   ServiceReference[] sr = null;
6   sr = null;
7
8   // Get producerId from the wireadmin
9   String producerId = (String) reference
10        .getProperty(org.osgi.framework.Constants.SERVICE_PID);
11
12  // Create wire to device
13  createWire(producerId, propertylistdispserv.getConsumerId());
14  return service;
15 }
```

However a `ServiceTracker` is useless when not being connected with an associated class that acts as a `Consumer` since a wire must ordinarily be created between a `Consumer` and a `Producer`. In this context we use a `Consumer` given by the reference called `propertylistdispserv` (See line 13 above). In order to make this possible we must first also associate the concrete `ServiceTracker` (in the code snippet above: `DeviceSTCustomizer`) with the `Consumer` instance. This is done with the following code lines.

```
1  // Register propertyupdate message dispatch service for clients
2  this.propertylistdispserv = new PropertiesUpdateDispatchImpl(bc,
3    dispatcher, "property.dispatch.service");
4
5  //...
6
7  // Initialize the tracker for the devices
8  if (customizerPropertyProvider == null)
9    this.customizerPropertyProvider = new DeviceSTCustomizer(bc,
10     propertylistdispserv);
```

Additionally we need the `dispatcher` (Line 3 above) that is as we will see later (See section: 11.3.5) one of the core classes in the communication subsystem that provides a broadcast mechanism which is used by the concrete `PropertiesUpdateDispatchImpl class`.

In this case the `PropertiesUpdateDispatchImpl class` provides an `update()` method (since it implements the `Consumer` interface) that is ready to receive new data across the wire as soon as the wire has been created by the dedicated `ServiceTracker`.
The code snippets taken from the `PropertiesUpdateDispatchImpl class` illustrate the `update()` method (Line 1) that invokes the `processProperties()` method (Line 7) that wraps the content of the wire object `(PropertyList)` in a new message provided by the communication subsystem. As soon as the message has been constructed with the appropriate message fields it will be broadcasted to all connected clients (Line 23).

```
1  public void updated(Wire excludeSourceWire, Object in)
2  {
3    try
4    {
5      if (in instanceof PropertyList)
6      {
7        this.processProperties((PropertyList) in, excludeSourceWire);
8      }
9
9      //...
10 }
11
12 //...
13
```

```
14 private void processProperties(PropertyList in, Wire excludeSourceWire)
15 {
16   // Create Return Message
17   // -1 therefore since we go straight down to the connection
18   Message msg = new SendMessage(-1, RBCConstants.MSG_PROPERTIESUPDATE);
19
20
21   // Add necessary fields to the message
22   //...
23   this.dispatcher.broadcastToClients(msg);
24 }
```

We complete this section with a sketch (See figure: 11.2) that illustrates each of the involved component and how they exchange information across the wires. Of course its kept very fundamental since we rather want to emphasize how the bundles and their services communicate to the server and vise versa. As one can depict the core exchanges a set of flavours (*PropertyList, ABIBusStatus, ABIDeviceStatus and AreaStatus*) (See section: 4.2.3.2)) which are intercepted by the server bundle that provides dedicated dispatch services each of which act as Consumer (such as the `PropertiesUpdateDispatchImpl class` mentioned above).

The virtual Producer will be explained after the communication subsystem since it's actually used by the handlers (See section: 11.3).

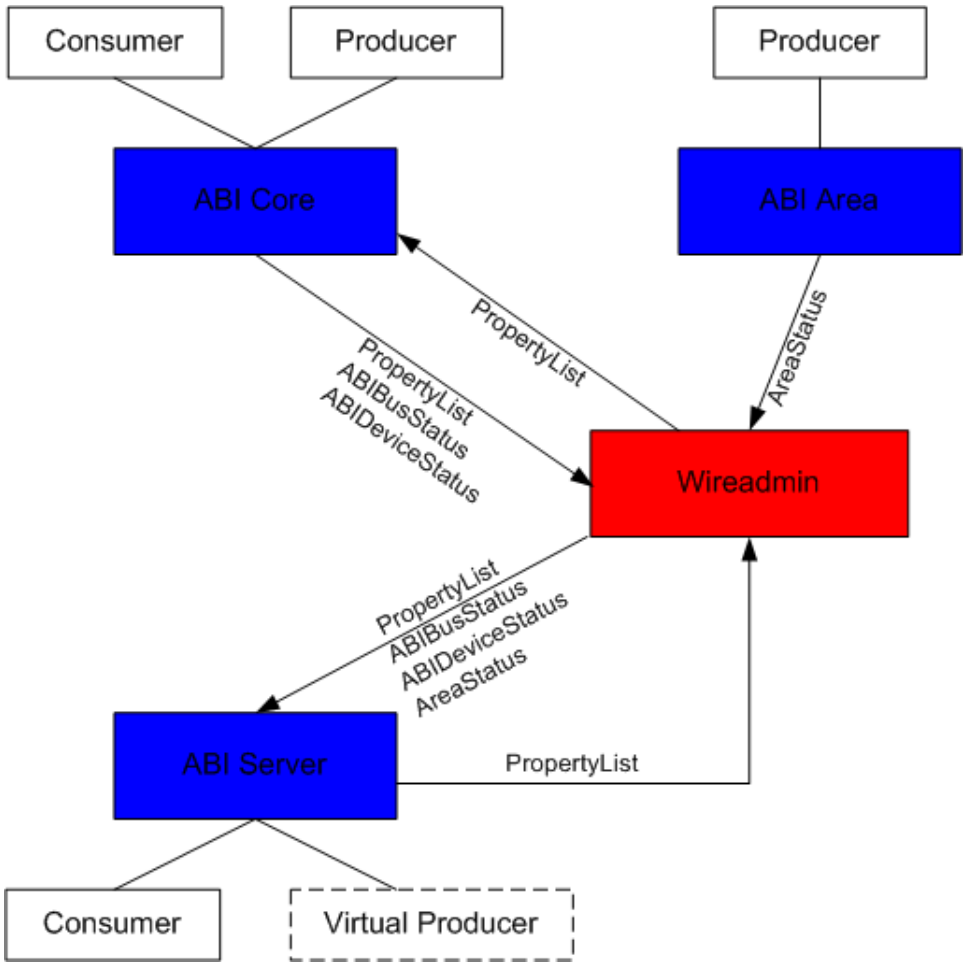


Figure 11.3: Wires to/from the RBC Server

11.3 Communication Subsystem

We recap that the communication subsystem is written to be reusable. In order to improve the re-usability, it has been split into multiple components that can be exchanged, if needed. The communication subsystem basically does all the low level part such as synchronous and asynchronous message calls and moreover it implements most of the features provided by the RBC Protocol Specification ([NB05b]).

The main components in the communication subsystem are:

- Connection Listener (See subsection: 11.3.1)
- Connection (See subsection: 11.3.2)
- Message (See subsection: 11.3.4)
- SendMessage (See subsection: 11.3.4)
- ReceiveMessage (See subsection: 11.3.4)
- Message Dispatcher (See subsection: 11.3.5)
- Message Handler (See subsection: 11.3.3)

For detailed information about the classes and their methods, see the source code snippets (below), the source code documentation ([JAVb]) and the UML class diagram (See figure: 11.3).

11.3.1 Connection Listener

The connection listener (Class name `ConnectionListener`) continuously listens on a specified TCP port for new connections. For each accepted connection, it creates a connection object that deals with the connection. It also has the ability to close all previously accepted connections on demand.

11.3.2 Connection

The connection class (Class name `Connection`) handles established connections. Each connection is represented by a Connection object. It continuously reads data from the remote peer and sends messages that were enqueued for sending. In order to provide a smooth program execution, each connection owns two worker threads (Reader and Writer).

The reader thread (Class name `ReaderThread`) is responsible for receiving messages that were sent by the remote peer. Whenever it has received a complete message, it forwards it to a message dispatcher (See figure: 11.3) that enqueues the message it its own receive queue (See section: 11.3.5).

The writer thread (Class name `WriterThread`) is responsible for sending messages to the remote peer. It reads messages that were previously enqueued in a send queue and sends them to the remote peer.

11.3.3 MessageHandler

The message handler (Interface name IMessageHandler) is only an interface that defines what methods a message handler must provide. The concrete message handler class is implemented outside the communication subsystem. Whenever a message arrives, the dispatcher forwards the message to the appropriate handler which does the *real* processing work. Additionally the dispatcher is also responsible for verifying the message.

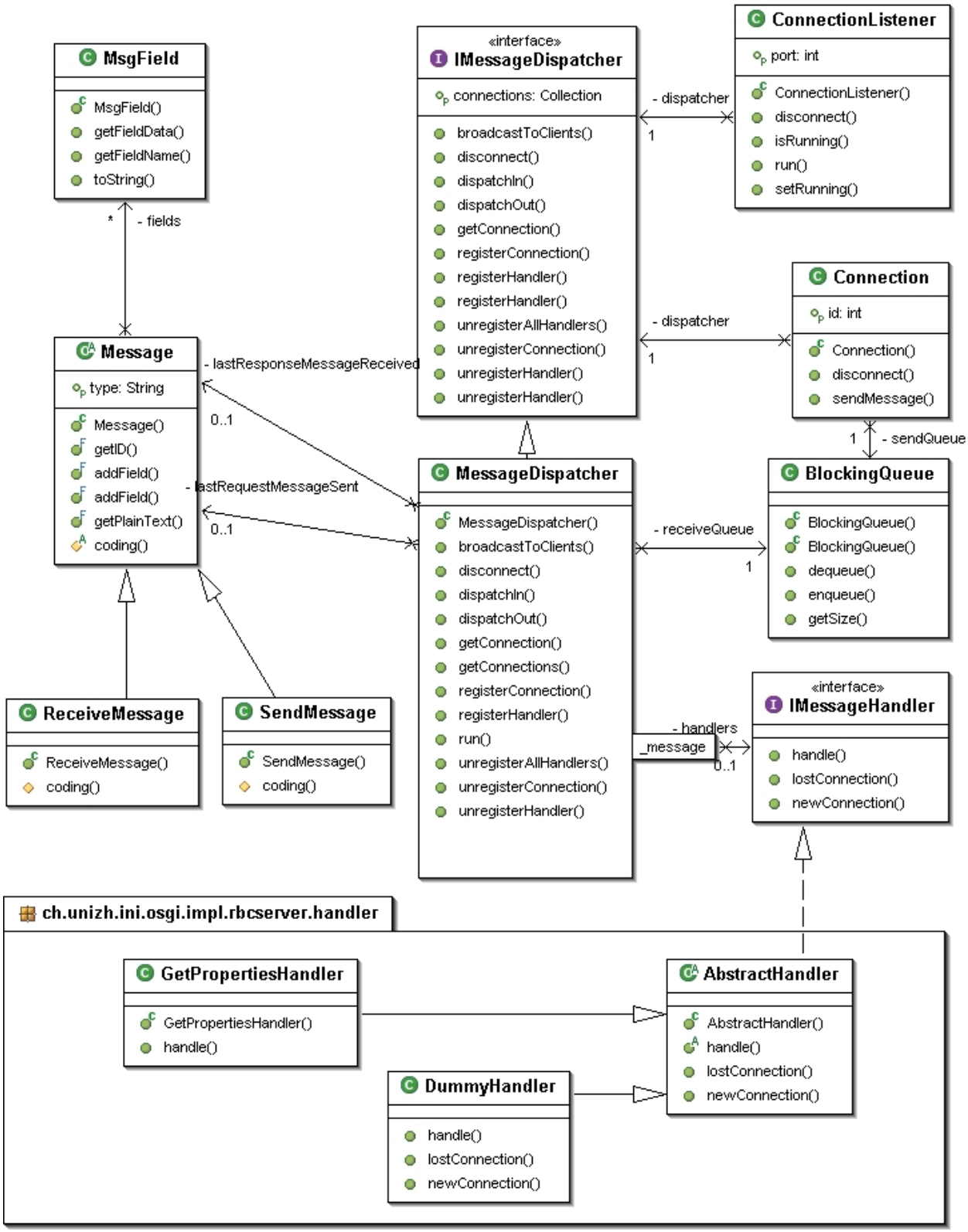


Figure 11.4: Communication Subsystem

11.3.4 Message

The message (Abstract Class name `Message)` represents a single message that can be sent to or received from a remote peer. It does all the required data encoding (if any) and provides methods for data access. Because we need to distinguish between incoming and outgoing messages this class provides two subclasses called `SendMessage` and `ReceiveMessage` which basically differ from each other only by its coding method. Therefore the `Message class` provides a template method called:

```
abstract protected StringBuffer coding(StringBuffer _data);
```

This method should be implemented by each of the two subclasses that provide a corresponding encoding and decoding schema.

11.3.5 IMessage Dispatcher and Message Dispatcher

Generally we can describe the concrete dispatcher or its interface with following words:
The message dispatcher (Class name `MessageDispatcher`) is responsible to distribute received messages to an appropriate message handler. It is also responsible for sending outgoing messages over the connection they were received on. A message handler must be registered with the message dispatcher before it can receive any message in the first place. Messages for which no handler has been registered are considered to be unknown and are required to be rejected according to the RBC Protocol Specification ([NB05b]).

The latter description was rather more abstract and hence need to be elaborated a bit because it does quite more then just that. According to the RBC Protocol Specification (See: [NB05b]) we must be capable of supporting synchronous and asynchronous message calls. Concretely speaking this means: With a blocking call any client will hang till the operation completes. In other words once invoked, the client will keep blocking till it gets the response (if any) from the server. This is a very useful technique when invoking ABI services (RSI's) that do not take long time to complete and the hanging in the client side is negligible. This will be a huge drawback in the client side performance, if the operation takes considerable amount of time. In contrast a non blocking call will provide the client to use a callback or polling mechanism to get the responses (if any) for a service invocation. Clients usually get this response it two ways. A good way, which has also been applied in the reference implementation of the RBC API ([NB05c]) is some sort of callback. Broadly speaking this means that almost each message type that a client or the server is interested in has a corresponding message handler that has been registered to a dedicated dispatcher. Herewith each message is processed by the competent handler upon dispatching.

How do we accomplish synchronous message calls? Truly speaking the server barely makes use of synchronous messages calls since it usually does not invoke services but rather respond to services. Nevertheless its crucial to be capable of supporting this feature as well.
The next following subsections actually would be more suitable to describe for the client API (RBCAPI) but since quite some code has been adapted from the server we rather prefer to describe it in here.

11.3.6 Message types

We recall that the RBC Protocol Specification ([NB05b]) distinguishes between three different type of messages: *Request, Response and Message.*
When performing blocking calls the clients creates a *Request* packet with an accompanying message-type. The Server will dispatch the message to the appropriate message handler that processes the message. When the server has completed all necessary tasks the server will send the appropriate *Response* message back to the client. Hereby the server alters the message with the message-type *response.* According to this schema we can fulfill the synchronous message calls in the client side.

When performing non-blocking calls the client simply needs to structure a message of type *message.* The server will then process the message by the corresponding callback handler that does the dedicated work

and will quite often provide a default common success response message (See: [NB05b]).

11.3.7 Synchronous messages calls

Well how do we accomplish synchronous message calls? The following code snippets should approximately demonstrated how this is accomplished.
Basically every time when sending a message one must define whether the message is required to be sent *asynchronously* or *synchronously*. Assuming that a message is requested to be sent synchronously (provided that the message type has previously been set to type *request*). Eventually the dispatch thread [1] will call up the `dispatchOut()` method (Line 1) when this new message is about to be sent (See section: 11.3.8).

The method therefore provides a way to block the current thread when sending a message synchronously. This of course is accomplished with a condition variable (Line 22-35) that simply blocks a specified amount of time until it will be released by the wait pool again (has either been released by the complementary `dispatchIn()` method or when the timeout has been reached). Please note that asynchronous messages don't need to provided this feature since they don't require to be blocked.

```
1  public synchronized Message dispatchOut(Message _msg, int _timeout)
2  {
3      Connection connection = null;
4
5      /**
6       * SYNCHRONOUS METHOD CALL typ:(Request-Response)
7       */
8      // Treat disconnect as a regular message if one inadvertently had
9      // set Message type request
10      if (_msg.getType().equals(Message.TYPE_REQUEST)
11              && _msg.getName().equals(RBCConstants.MSG_DISCONNECTCLIENT) == false)
12      {
13          connection = (Connection) this.connections.get(new Integer(_msg
14                  .getID()));
15
16          if (connection != null)
17          {
18              connection.sendMessage(_msg);
19              lastRequestMessageSent = _msg;
20          }
21          hasArrived = false;
22          // Block Thread until response
23          while (!hasArrived)
24          {
25              try
26              {
27                  if (timedwait(_timeout)==false)
28                  {
29                      return null;
30                  }
31              }
32              catch (InterruptedException e)
33              {
34                  return null;
35              }
36          }
```

[1] For the simplification we assume that the dispatch thread calls up the `dispatchOut()` method directly although this isn't true (See section: 11.3.8 where we use a `ThreadPool` instead)

```
37          return lastResponseMessageReceived;
38      }
39      /**
40       * ASYNCHRONOUS METHOD CALL (type: Message)
41       */
42      else
43      {
44          connection = (Connection) this.connections.get(new Integer(_msg
45                  .getID()));
46
47          if (connection != null)
48          {
49              connection.sendMessage(_msg);
50          }
51      }
52      return null;
53  }
```

The method `dispatchIn()` is called by the `ReaderThread` when a new message is considered as completed. Hence we need to define whether messages have been sent asynchronously or synchronously. Line 6-9 accordingly check if we've received our corresponding return message. Assuming that we've been waiting on a response message (synchronous call) that ultimately has been received by the reader thread that will accordingly notify the dispatch thread about the reception. The dispatch thread can then wakeup and proceed and finally return the message.

```
1  public synchronized void dispatchIn(Message _msg)
2  {
3      try
4      {
5          // Waiting request to satisfy?
6          if (_msg.getType().equals(Message.TYPE_RESPONSE)
7                  && _msg.getName().equals(RBCConstants.MSG_REJECT)
8                  || _msg.getType().equals(Message.TYPE_RESPONSE)
9                  && _msg.getName().equals(lastRequestMessageSent.getName()))
10         {
11             /**
12              * SYNCHRONOUS METHOD CALL completed typ:(Response)
13              */
14             hasArrived = true;
15             lastResponseMessageReceived = _msg;
16             notifyAll();
17         }
18         else
19         {
20             receiveQueue.enqueue(_msg);
21         }
22     }
23     catch (QueueFullException e)
24     {
25         e.printStackTrace();
26     }
27 }
```

Please note that the RBC Server usually does not use synchronous method calls since they are usually not order of the day but rather need to support a set of asynchronous message handlers that process the message and frankly speaking would actually only require to distinguish whether each received message is of type

message or *request*. However since we want to provide an entire subsystem that can easily be plugged and extended the servers (RBC Server, Remote Service Server) as well as the client (RBC API [NB05c]) basically provide more or less the same subsystem.

11.3.8 Dataflow summary

When the connection listener registers (See section: 11.3.1) an incoming connection, it creates a new connection object that handles the connection from this point. The new connection object gets the message dispatcher (See section: 11.3.5) assigned to it by the connection listener. Hereby the message dispatcher keeps a list of all active connections which allows a simple termination of all connections.

Messages that need to be sent are passed to the message dispatcher which checks whether the message needs to be sent synchronously or asynchronously. Asynchronous message are enqueued in a send queue and later dequeued by the corresponding writer thread of the connection that sends the messages to the remote peer. One further note is that messages are sent in the same order as they were enqueued.

Prior to receiving messages, message handlers must be registered with the message dispatcher. Each handler can register one or more messages it wishes to process. The reader thread of a connection object continuously reads messages from the input stream. As soon as a complete message is received, it forwards the message to the message dispatcher (associated with the connection) by calling `dispatchIn()`. The message is hereby enqueued in a receive queue provided by the message dispatcher.
Correspondingly the entire dispatcher provides its own thread that processes the messages but similar it does not execute any handlers within its context since we don't want to block the dispatch thread from dispatching new enqueued messages either. So we considered a ThreadPool which does that kind of work.

In particular the dispatch thread continuously dequeues (Line 7) from the receive queue, roughly process the message header and will then excavate any handler tasks to a separate thread in a `ThreadPool`, in order to shorten the processing time needed by each handler. Thus assisting the dispatch thread in the form that the dispatch thread can return to the `BlockingQueue, dequeue()` method quite faster and hereby enhance the speed of dispatching work drastically. One of the competing threads in the ThreadPool can then process the incoming message by its correpondig handler (Line 19).
This was necessary since the two thread exchanges can really boost performance. The following code snippets illustrates that.

```
1 public void run()
2 {
3     while (isRunning && Thread.interrupted() == false)
4     {
5         //...
6
7         msg = (Message) this.receiveQueue.dequeue();
8
9         //...
10
11         // Get corresponding message handler from hash-map
12        handler = (IMessageHandler) handlers.get(msg.getName());
13
14        //...
15
16        // Dispatch to corresponding handle for processing...
17        // Might return a new Message (Response) depending on
18        // the message type
19        threadpool.runTask(new HandlerThread(handler, msg));
20    }
21 }
```

11.3.9 Extendability

In the beginning of this section we quoted that the subsystem is easily expendable. Assume that you want to add some more functionality to your ABI System. Some functionality that that has not been provided by the ABI System yet. According to our subsystem you need to do three things.

1. The first one is rather more a formal thing but all other developers will appreciate it if you comply to this rule: Please keep track of all defined messages in the RBC Protocol ([NB05b]) We assume that you want to add some functionality to the ABI System. Moreover you might have already done that but you want to make it accessible for distributed client applications. First read up on the RBC Protocol Specification ([NB05b]) define and add your custom message to the protocol and document it. Please do never delete any fields of existing messages! But rather use the fields deprecated instead!
   Then add your definition of the message to the file called `RBCConstants.java` also. Herewith you have defined a new message and can proceed to the handler implementation.

2. As indicated in point one you will then have to implement you own message handler that should process your message. An example of such a hander is given below. Note that you might want to inherited from the `AbstractHandler` (Line 1) as well since you might don't want to implement other methods which would be necessary when implementing to the interface `IMessageHandler` directly (See JavaDoc: [JAVb]).

```
1  public class DummyHandler extends AbstractHandler
2  {
3    public Message handle(IMessageDispatcher _dispatcher, Message _msg)
4    {
5       LinkedList list = null;
6       MsgField msgField = null;
7
8       // Read message content
9       list = _msg.getFields();
10
11      while (list.isEmpty() == false)
12      {
13         try
14         {
15            msgField = (MsgField) list.removeFirst();
16            System.out.println("Handler: " + msgField.getFieldName()
17                     + ":" + msgField.getFieldData());
18         }
19         catch (NumberFormatException e)
20         {
21            e.printStackTrace();
22         }
23      }
24      return null;
25   }
26 }
```

3. After having finished the hander code you need to register your handler to the MessageDispatcher (See section: 11.3.5) by invoking the `registerHandler()` method (Line 11). In the RBC Server bundle this is achieved in the `Activator class`.

```
1  private void startListen() throws IOException
2  {
3      //...
```

```
4
5       /**
6        * Register supported handlers...
7        * More specific information about each message-type or handlers
8        * can be obtained from: the Handlers, RBCConstants
9        */
10
11      this.dispatcher.registerHandler(RBCConstants.YOURMESSAGENAME,
12        new YOURMESSAGEHANDLER(dispatcher));
13
14      //...
15
16      /**
17       * Set the server into listening-Mode
18       */
19      listener = new ConnectionListener(1234, dispatcher);
20      listener.start();
21  }
```

This is it! You can check on it by using `telnet` and typing your message format in it.

11.4 Virtual producer

We might have used the term `Virtual producer` quite some time but we haven't really discussed him yet. Truly speaking the `Virtual producer` is actually a feature provided by the ABI Core and is not part of the RBC Server. However we think that it is more suitable to discuss him now the in the core since the server is momentarily the only component that actually makes use of the `Virtual producer`. The `Virtual producer` basically creates a temporary wire that is created upon calling the `send` method. A virtual producer is necessary since quite often no long lasting wires are required. i.e. When switching off the lights. But according to the the wireadmin concept we span a wire just to deliver the message. As soon as we completed the delivery we kill the wire right away though.
Don't get confused about this concept since it has nothing to do what we've discussed in section 11.2. Section 11.2 covered the wires that are required when the server is acting as Consumer since it must receive and broadcast for instance changing properties to distributed clients. The question is though how about the opposite? How can we deliver messages from the server to the devices? To tell the truth we only use the `Virtual producer` in the `SetPropertiesHandler class`. Since we work with properties and properties are provided by the `PropertyProvider` that implements the `Consumer` as well as the `Producer` interface we use the `Wireadmin` to deliver the message instead of using interfaces directly (See second code snippet).

Areas in contrast do not implement a `Consumer` interface and hence are not capable of receiving message the opposite way through wires. This makes sense since we don't require any properties in there. So in this case we consult the interfaces instead (See first code snippet)

With interfaces: (Code snippets from the `CreateAreaHandler class`)

```
1  String filter = "(objectclass=" + AreaServiceManager.class.getName() + ")";
2
3  ServiceReference references[] = null;
4  try
5  {
6      references = bc.getServiceReferences(null, filter);
7  }
8  catch (InvalidSyntaxException e)
9  {
```

```
10     e.printStackTrace();
11 }
12 // Retrieve the AreaServiceManager object
13 this.areaServiceManager = (AreaServiceManager) bc
14   .getService(references[0]);
15
16 //...
17 if (this.areaServiceManager.createAreaService(areapid, location))
18 {
19   //...
20 }
21  //...
```

Using a `Virtual producer`: (An excerpt from the `SetPropertiesHandler class`)

```
1  // Set the properties
2  Properties props = new Properties();
3  props.put(org.osgi.framework.Constants.SERVICE_PID,
4     VirtualProducer.NAME);
5
6  // Define the flavours
7  props.put(WireConstants.WIREADMIN_PRODUCER_FLAVORS,
8         new Class[] { PropertyList.class });
9
10 String[] deviceClazzes = { Producer.class.getName(),
11    VirtualProducer.class.getName() };
12
13 // Create the Virtual Producer
14 VirtualProducer vpservice = new VirtualProducer(bc, _deviceurl);
15
16 reg = bc.registerService(deviceClazzes, vpservice, props);
17 vpservice.setServiceRegistration(reg);
18
19 // Send list using the Virtual Producer
20 vpservice.send(_propertylist);
21 // deregister virtual command device
22 vpservice.deregisterDevice();
```

Chapter 12

ABI Remote Service Bundle

According to the "notes" that has been provided in section: 4.2.4.4 we keep this chapter as low as possible since it's still being tested and elaborated.

12.1 Overview

Anyway since the design won't change we can illustrate it in figure: 12.1

The two core classes in this subsystem are evidentially the `RemoteBusServiceImpl` and the `RemotePresenceServiceImpl` classes. Because of the fact that any device whether or not it provides a virtual one, requires a new concrete bus implementation to be programmed, we need to preserve this concept in order to be compatible with the ABI core (See chapter: 8). Hereby the `RemoteBusServiceImpl class` takes over leadership by providing its own server that actively polls the presence status on each *PC Presence Device* (Personal Computer). This has been done because we need a way to figure out if a *PC Presence Device* has been either terminated (means the a user PC has been shut down) or the *PC Presence Device* has suddenly become unreachable. Hence by letting the server poll for each client we can simply initiate a synchronous procedure call by sending the message: `getpresence` (See appendix: A.1.3.2) that retrieves whether each client (user) is present or not.

Naturally we must be capable of identifying each *PC Presence Device* upon a new connection establishment. Concretely speaking this means that every *PC Presence Device* must have an unique id. Thus we need a way to find out whether the user has already been assigned an unique id or not. To solve this issue we distinguished following two cases:

1. The *PC Presence Device* doesn't posses an unique id and is therefore also unknown to the Remote Server and initiates a regular synchronous `connect` message (See appendix: A.1.3.1).
   The server will then try to register the *PC Presence Device* as a new virtual device and provide an appropriate return message with the newly generated unique id enclosed.
   The client will then need store the received id for any possible re-identification purposes.

2. The *PC Presence Device* already owns an unique id and hence initiates a `connect` message but this time with the stored id enclosed. Herewith the server is capable of recognizing and verifying the *PC Presence Device*'s re-existence. Equally to the latter case we subsequently create a new virtual device with the help of the provided id and of course acknowledge it back to the remote client in form of a return message.

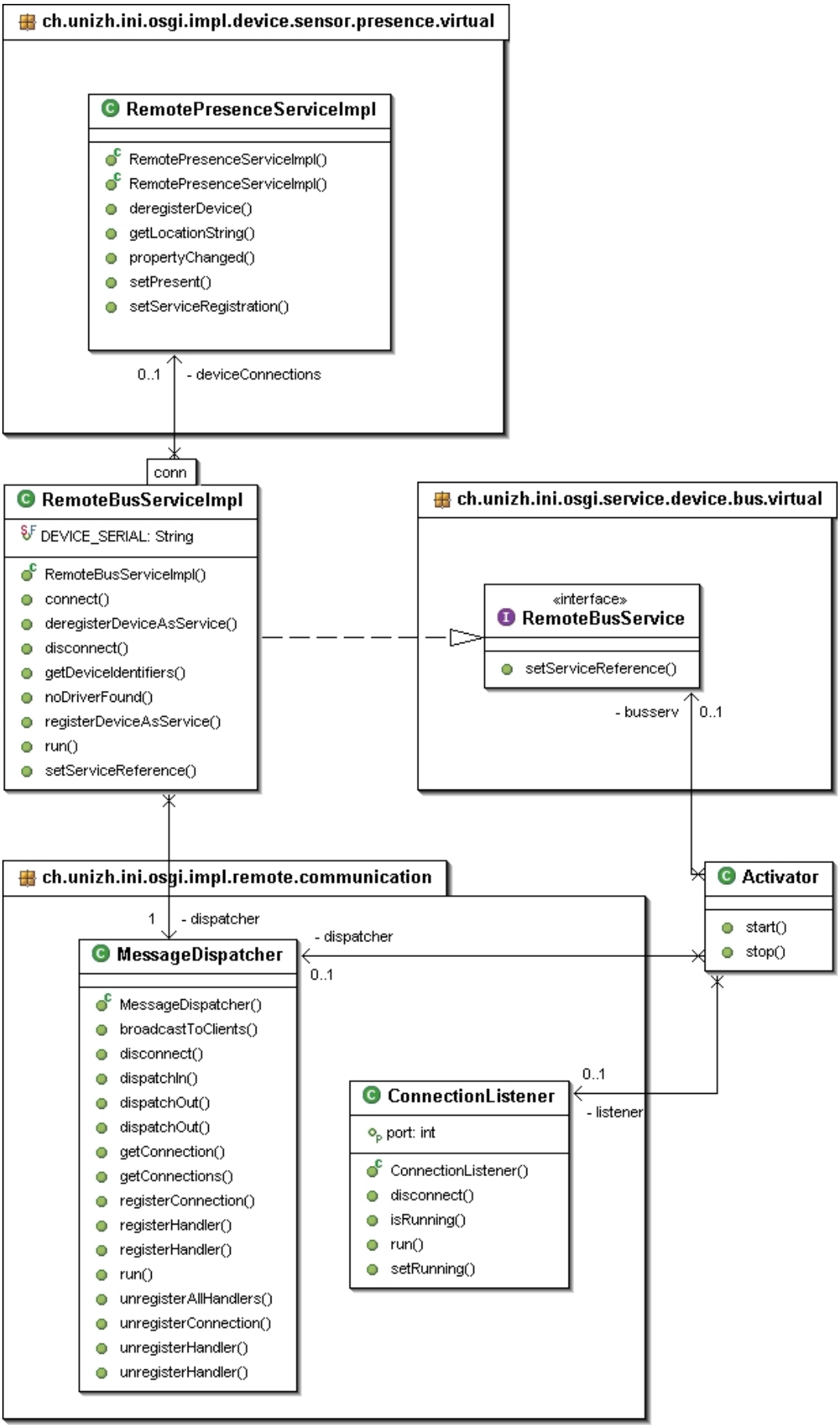


Figure 12.1: Virtual Bus

12.2 Further notes

Communication Subsystem: The communication subsystem has been more or less adapted from the RBC Server. So for more information please consider other sections within the chapter: 11 for this instead.

remotePresenceServiceImpl: We skip any further explanation about this class since we assume that you should have seen how this works by now. So for more information about how to use properties or how to implement a new device consider following sections: 8.5.2, 9.2.1. You might also want to have a look at the appropriate JavaDoc section: [JAVb] or even the source directly.

Chapter 13

ABI Discovery Service Bundle

This chapter has just been added since it has been implemented within a separate bundle. But isn't actually worth to be discussed at all because its kept very fundamental. In short we present the simple UML diagram (See figure: 13.1).

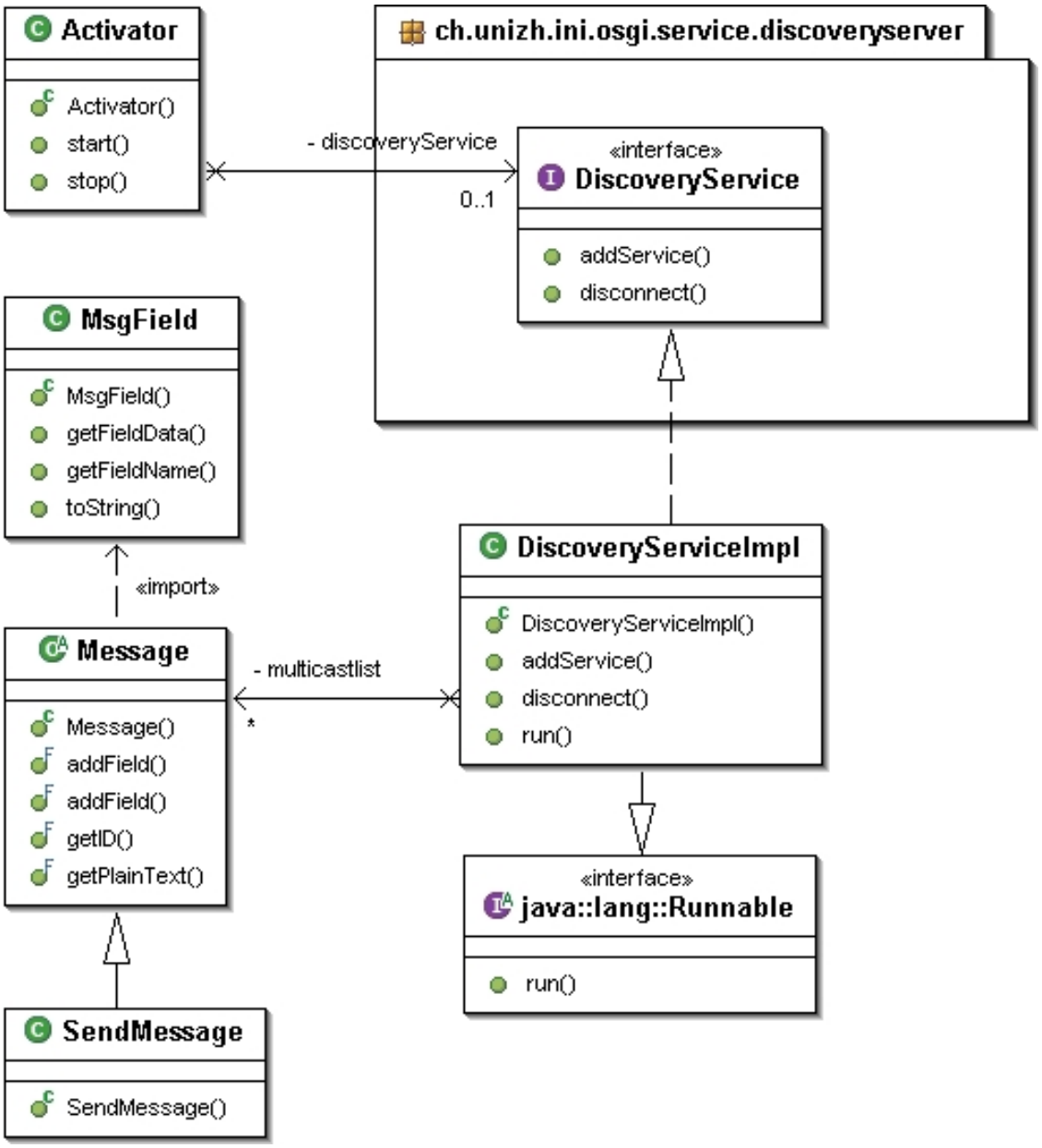


Figure 13.1: Discovery Server Bundle

Basically this bundle is intended to provide a discovery service that can be used by bundles which provide a dedicated server that might benefit from this service in form that they can assign a personal `ServiceTracker` (See [OSG]) which can track down the the availability of the discovery bundle.
Basically they would need to check whether this bundles is up and running or not. When the discovery bundle has been plugged into ABI System they might consider of using it by simply invoking the method called: `addService()` where they can make their service public.

Herewith we support any client whether it's a distributed client application or even a remote service application such as the *PC Presence Service* by providing one global multicast address and group. Thus no client does really need to know about any service provided by the ABI System directly. A further benefit is that we can provide low coupling in the sense that one could change all server ports to any port they require to run completely independent of any client application. Furthermore we don't need to put up with dynamically changed ip addresses anymore (DHCP) which frankly speaking does happen if it has not been statically assigned before.

In order to receive any multicast message you need to listen on port **6000** and join the appropriate multicast group with the following address: **225.4.5.6**.

The message you will be receiving has been defined in the appropriate appendix section: A.2.

Part V

Appendix, Glossary and Bibliography

Appendix A

Appendix

A.1 Remote Service Protocol

A.1.1 Introduction

The Remote Service Protocol (RSP) is an application layer protocol for distributed, collaborative, hypermedia information systems. The format of RSP describes a generic, stateless, protocol which can be used for many tasks beyond its use. Its basically an extenuated version of the RBC Protocol [NB05b]. So in order to acquire more information please consult this documentation.

The purpose of the RSP Protocol is to standardize the communication between the ABI Remote Service Server and remote services (See section: 4.2.4).

A.1.2 Defined messages

All fields which are currently specified by this protocol are now given regardless of their message type (request, response or message) .
Further please note that we don't cover any message which has already been defined by the RBC Protocol [NB05b]. So for such messages please consult the RBC Protocol specification.

```
messagename   =  reject            (RBC Protocol)
               | peer-id           (RBC Protocol)
               | disconnectclient  (RBC Protocol)
               | connectpresence
               | getpresence
               | setpresence
```

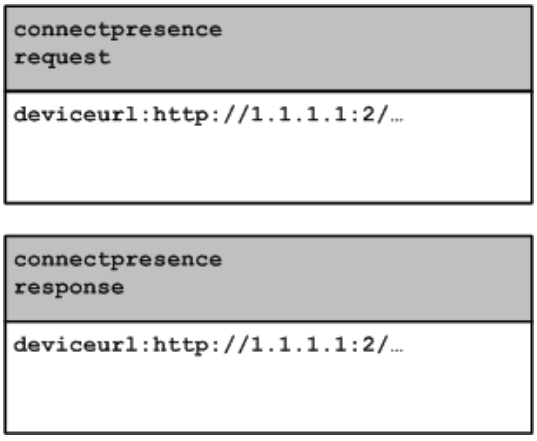
A.1.3 Remote Presence Service Message Packets

A.1.3.1 connectpresence message

Message-Name: connectpresence
Scope: Client → Server
Common Message-Type: REQ → RES
Protocol Version: 1.0
Deprecated since: N/A
Description: The connectpresence command connects a virtual presence device to the remote server
Body parameters:

1. REQ[1], deviceurl: The unique device identifier i.e. http://1.1.1.1:2/...
2. RES, deviceurl: The unique device identifier i.e. http://1.1.1.1:2/...

Message example:



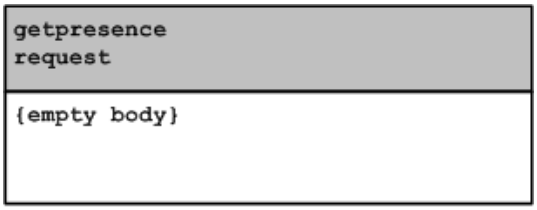
[1] This parameter is optional and used for identification purposes

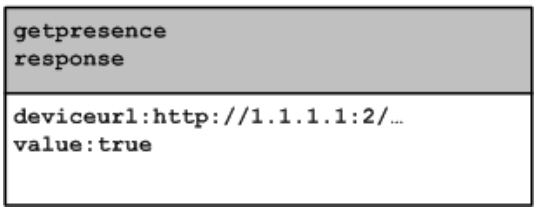
A.1.3.2 getpresence message

Message-Name: getpresence
Scope: Server → Client
Common Message-Type: REQ → RES
Protocol Version: 1.0
Deprecated since: N/A
Description: This command returns either true or false (according to the presencestatus of the called client)
Body parameters:

1. REQ, {empty body}
2. RES, deviceurl: The unique device identifier i.e. http://1.1.1.1:2/...
3. RES, value: {true, false}

Message example:





A.1.3.3 setpresence message

Message-Name: connectpresence
Scope: Client → Server
Common Message-Type: Message
Protocol Version: 1.0
Deprecated since: N/A
Description: The setpresence command notifies the server about a new presencestatus change
Body parameters:

1. deviceurl: The unique device identifier i.e. http://1.1.1.1:2/...
2. value: {true, false}

Message example:

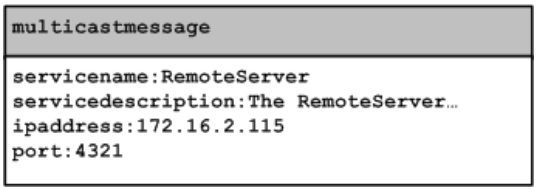
A.2 Discovery Service Protocol

The discovery message has been defined as follows:

Message-Name: multicastmessage
Scope: ABI Discovery Bundle → Remote Service Device(Client)
Common Message-Type: Message
Protocol Version: 1.0
Deprecated since: N/A
Description: The multicastmessage frequently broadcasts ABI System information such as available servers, etc.
Body parameters:

1. servicename: The name of the service. i.e. RemoteServer
2. servicedescription: A detailed description of the server
3. ipaddress: The ip address of the server
4. port: The port of the server. i.e. 1234

Message example:



Appendix B

Glossary

Abbreviation	Explanation / Comment
ABI	Adaptive Building Intelligence
ABI System	The ABI System implemented using an OSGi Framework
AI	Artificial intelligence
Apache Axis	Apache Axis is an implementation of the SOAP
API	Application Programming Interface
Bundles	Java JAR archive
DAI	Distributed Artificial Intelligence
DHCP	Dynamic Host Configuration Protocol
DSP	Discovery Service Protocol
ETH	Swiss Federal Institute of Technology
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEEE 1284	IEEE 1284 is a standard that defines bi-directional parallel communications between computers and other devices. It is most frequently used for connecting computers to printers but it has also been used to connect storage devices, for file transfer and remote access between computers, and to communicate with modems ([WIK]).
INI	Institute of Neuroinformatics
JAR	Java Archive
LNS	Lon Network Server
LonWorks	Field bus network protocol / standard
OSGi	Open Service Gateway initiative
RBC	Remote Building Control (Protocol)
RBC Server	The Server Bundle of the ABI System
RBC API	The protocol abstraction layer API that implements the RBC Protocol
REQ	Refers to a message request
RES	Refers to a message response
RFC	Request for Comments
RMI	Remote Method Invocation (Java Technology)
RSI	Remote Service Invocation
RSP	Remote Service Protocol
RS232	RS-232 (sometimes also referred to as EIA RS-232C) is a standard for serial binary data

Abbreviation	**Explanation / Comment**
Service	Interface exported by the service provider bundle
SOAP	Simple Object Access Protocol (W3C)
UML	Unified Modeling Language
UNIZH	University of Zurich, Switzerland
XERCES	Advanced XML Parser with many supported features

Bibliography

[AG05] Feller AG. Falcon device protocol. Technical report, Feller AG, 2005. 4_Frames_Ae2.xls.

[ALT] Alternating bit protocol. http://en.wikipedia.org/wiki/Alternating_Bit_Protocol.

[Apa] Web services - axis. http://ws.apache.org/axis.

[BG04a] Patrick Brunner and Simon Gassmann. Adaptive building intelligence based on the open services gateway initiative, diploma thesis. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2004.

[BG04b] Patrick Brunner and Simon Gassmann. Adaptive building intelligence based on the open services gateway initiative, term work. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2004.

[EAS04a] Lprs data sheet – easy-radio er900ts transmitter, er900rs receiver & er900trs transceiver, 2004. 1_EasyRadio900RTx.pdf.

[EAS04b] Lprs data sheet – easy-radio demonstration kit & programming software, 2004. 3_EasyRadiokit.pdf.

[EAS04c] Lprs data sheet – easy-radio guide, 2004. 2_Easy-Radio Software Guide 1-3.pdf.

[FAL] Falcon applications.

[HTT] Rfc 2616 (rfc2616). http://www.faqs.org/rfcs/rfc2616.html.

[JAVa] The java communications api. http://java.sun.com/products/javacomm/index.jsp.

[JAVb] Rbc api javadoc.

[KNO] Knopflerfish. http://www.knopflerfish.org/index.html.

[NB05a] Stephan Kei Nufer and Mathias Buehlmann. Adaptive building intelligence – administration tool, 2005.

[NB05b] Stephan Kei Nufer and Mathias Buehlmann. Remote building control protocol – a protocol that is used to transfer building data. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2005.

[NB05c] Stephan Kei Nufer and Mathias Buehlmann. Remote building control (rbc) api – an api that supports custom agent application to remotely control a building. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2005.

[OPT05] Optfilter, 2005. 8_OptFilter.xls.

[OSG] Osgi alliance. http://www.osgi.org.

[RJD04] U. Rutishauser, J. Joller, and R. Douglas. Control and learning of ambience by an intelligent building. *IEEE Transactions on System,Man and Cybernetics Part A, Submitted*, 2004.

[RS02] Ueli Rutishauser and Alain Schaefer. Adaptive home automation – a multi-agent approach. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2002.

[Sch04] U. Schlegel. Expsyst_beschreibung, 2004. ExpSyst_Beschreibung.doc.

[TAO01] Tsl250r, tsl251r, tsl252r – light to voltage optical sensors, 2001. 7_TAOS_TSL250R.pdf.

[TAO03] Sharping the future of lightsensing solutions, 2003. 6_TAOS_brochure.pdf.

[TZ03a] Jonas Trindler and Raphael Zwiker. Adaptive building intelligence – an approach to adaptive discovery of functional structure. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2003.

[TZ03b] Jonas Trindler and Raphael Zwiker. Adaptive building intelligence – parallel fuzzy controlling and learning architecture based on a temporary and long-term memory. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2003.

[VIS99] Srn-2000c/pc serie – installationsanleitung, 1999. 5_VISONIC_SRN2000.pdf.

[WIK] Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Main_Page/.

[WIR05] Wireless battery powered presence detector, 2005. 9_Wireless PD.pdf.

[XER] Xerces java parser 1.4.4 release. http://xml.apache.org/xerces-j/.

Index