RBC API

Remote Building Control API
an API that supports custom agent application to remotely control a building

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

HSR
UNIVERSITY OF APPLIED SCIENCES
RAPPERSWIL
COMPUTER SCIENCE

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

uni | eth | zürich

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

Compiled: February 7, 2006

Typeset by LaTeX

Preface

by Nufer Stephan Kei and Buehlmann Mathias

This documentation is part of the term project and has been used in the ABI Admin application[NB05a] and partly also in the RBC Server [NB05b]. The underlying protocol specification which has been implemented by this API is called RBC Protocol [NB05c]. This API will play a central role in our upcoming diploma thesis.

This document provides additional aid when developing custom client applications for the ABI System. It explains the concepts that have been applied and most importantly it provides examples on how to use this API in order not to lose valuable time in any future development activities.

We hope that it helps. Good luck!

Stephan and Mathias.

Abstract 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 ([NB05c]).

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 (See chapter: 9) or can even provide an entire building intelligence.

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).

Table of Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Overview

The purpose of this document is to describe the API to be used when developing custom ABI client applications.
The benefit of such an API is that any ABI client application does not depend on the complex ABI system anymore. Instead any ABI client can now take over the control of the ABI System in a distributed fashion. In order to accomplish such an API a protocol needed to be developed and standardised. ([NB05c]) The API hereby supports most of the features that the protocol prescribes.

1.2 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. More details about each specific implementation see section: 4

1.3 Document structure

This document is structured into six parts: The Introduction (See part: I), Analysis (See part: II), Architecture, Design and Implementation (See part: III), RBC API Tutorial (See part: IV), Discussion and Future Works (See part: V) and the Glossary and Bibliography (See part: VI).
The second part (See part: III) gives a general introduction about the context we are dealing with. For this purpose we introduce a short domain model (See figure: 2.1) that abstracts the domain of application.
In chapter 3 we then introduce the architecture and the design of the API. You will see how the API has been structured and collaborate with its defined components.
One of the core elements of this article then builds the RBC-Tutorial (See part: IV) that first guides you though a short introduction in how to start the ABI System (See chapter: 6), how to install and start the necessary bundles that are part of the ABI System and most importantly how to setup the RBC API with your environment (See section: 6.2) such as the Eclipse Platform ([ECL]). Right after that you will be accompanied to implement your first distributed RBC client application (See chapter: 7). Chapter 8, 9 and 10 will then cover some more examples and will leave you with a Discussion and Future Work (See part: V) that clearly recaps addressed and solved issues and also forecasts what needs to be done in the upcoming diploma thesis (See section: 11.3).

Part II

Analysis

Chapter 2

Analysis

2.1 Introduction

The domain analysis in our context recalls the needed core elements that are needed to acquire all necessary building information (knowledge) such as presence and daylight sensors, etc. which are applicable when developing any intelligence that should be capable of controlling a building.
In order to get to know what kind of logic we are dealing with we cover following subjects in the analysis before moving on to the architecture and the design.

- A Domain Model that illustrates the statical view
- Necessary dynamics (current environmental data)
- Coverage and dependencies of the protocol specification
- API abilities and responsibilities (Needs)
- API Goals

2.2 Domain Model

The core intention of the API is to provide a reproduction of the complex ABI System in a small kind of form. For this purpose we developed a small domain model that reflects the environment in which any further development activities may take advantage of.
The concepts used in the domain model won't need any further explanation since the names of the concepts speak for themselves. (See figure: 2.1)

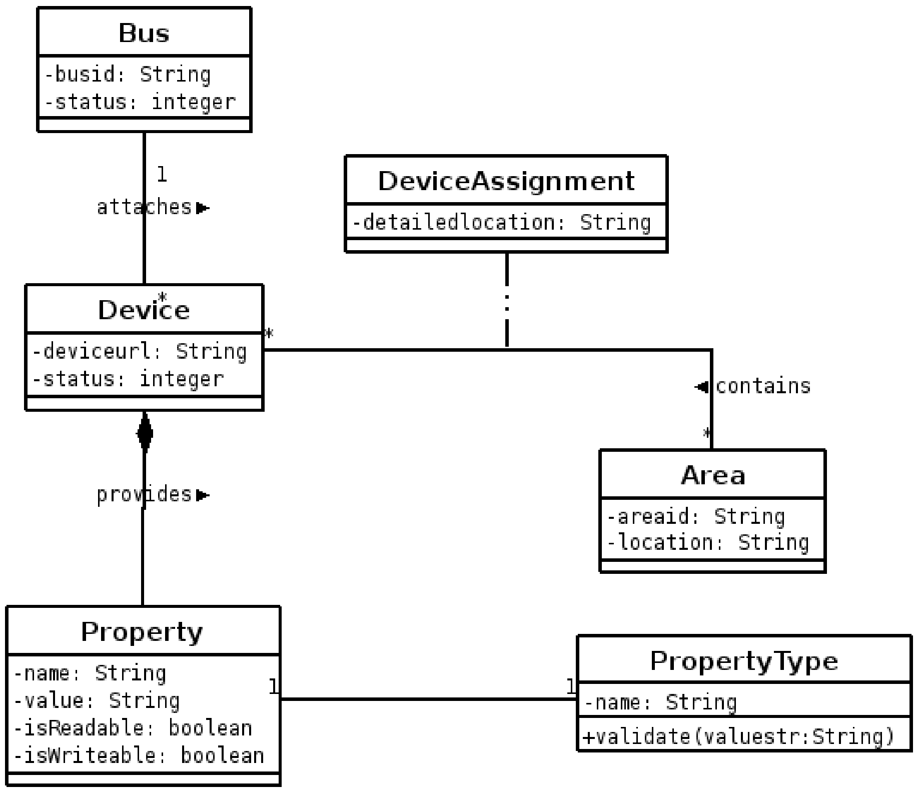


Figure 2.1: Domain Model

2.3 Dynamics

One of the issues we got confronted with was that devices quite often provide different settings and possibilities to either query them or not. This means that each device must provide and determine its capabilities in form of properties ([NB05b]). For instance: A Falcon light devices lighttstatus can be set and does even provide a message that is being sent right after each status change. On the other hand we can not presume such a feature as a requirement for each single device.
This is one major restriction we need to conform and to consider. An other aspect we need to watch out for is that upon any user interaction i.e. light status change must be recognizable by other clients as well. To counteract this issue the server has been implemented using a broadcast mechanism that informs any clients about that incident. In a nutshell we can summarize following criteria.

- hardware can be set (means i.e. A light can be switched on)
- hardware can provide status information such as announcing presence
- the API must be capable of receiving any changing state held by the server.

Basically we can distinguish between three cases that we need to think of when designing the API. These are illustrated in figure: 2.2, 2.3 and 2.4.
We have incoming messages that we classify as update messages. (See table: 2.1). So we need to be capable of receiving such messages from the server. In this case the API must be provided with all possible update messages prescribed by the protocol specification ([NB05c]).

Update message	Description
busupdate	Update messages which contain information about a specific bus such as busid and the state of the bus.
deviceupdate	Update messages which contain information about newly registered or deregistered devices.
propertiesupdate	Update messages which contain properties of a device.
areaupdate	Update messages which contain information about created and deleted areas, registered devices, their detailed location, etc.

Table 2.1: Update messages

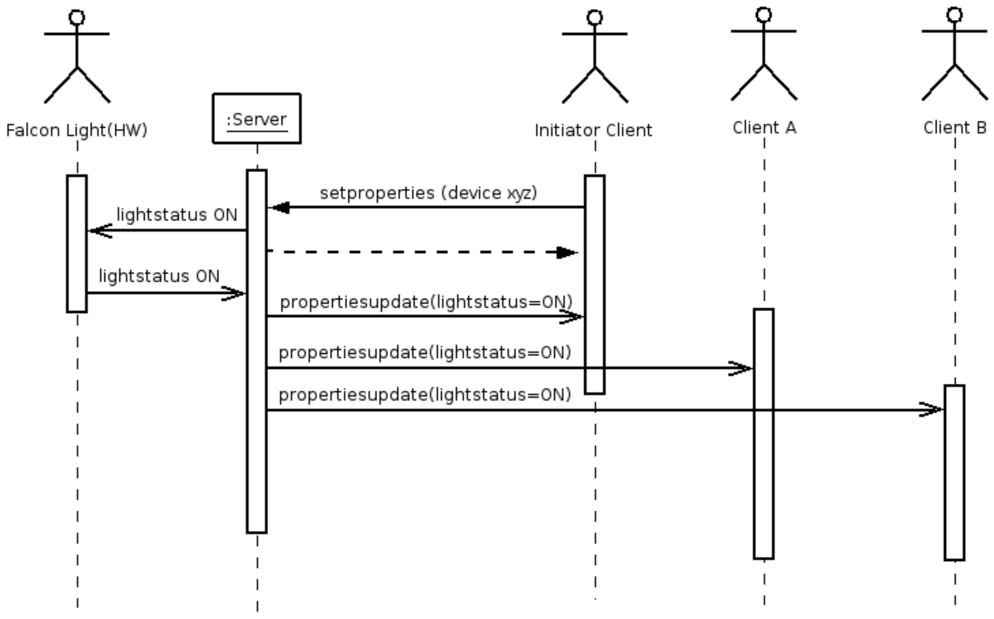

Falcon Light(HW)
:Server
Initiator Client
Client A
Client B
setproperties (device xyz)
lightstatus ON
lightstatus ON
propertiesupdate(lightstatus=ON)
propertiesupdate(lightstatus=ON)
propertiesupdate(lightstatus=ON)

Figure 2.2: Initiated property change

In this sequence diagram (See figure: 2.2) we can observe that any client can initiate a device property change (Provided that this device can be changed). All participants must be notified that the properties have been altered, and accordingly must be adapted by all client platforms in order to preserve consistency.

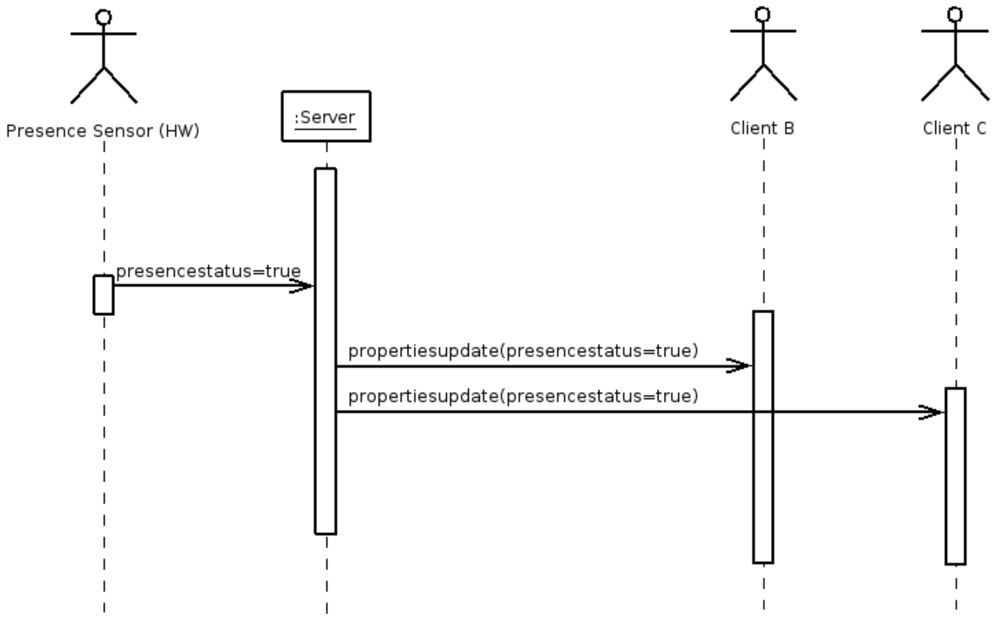

Presence Sensor (HW)
:Server
Client B
Client C
presencestatus=true
propertiesupdate(presencestatus=true)
propertiesupdate(presencestatus=true)

Figure 2.3: Autonomous property change

On the other hand when devices provide the capabilities to signal any change of state to the server this messages must be forwarded to any available stations. Analogically all participants must be notified about the signal in order to enable consistency on each client platform (See figure: 2.3).

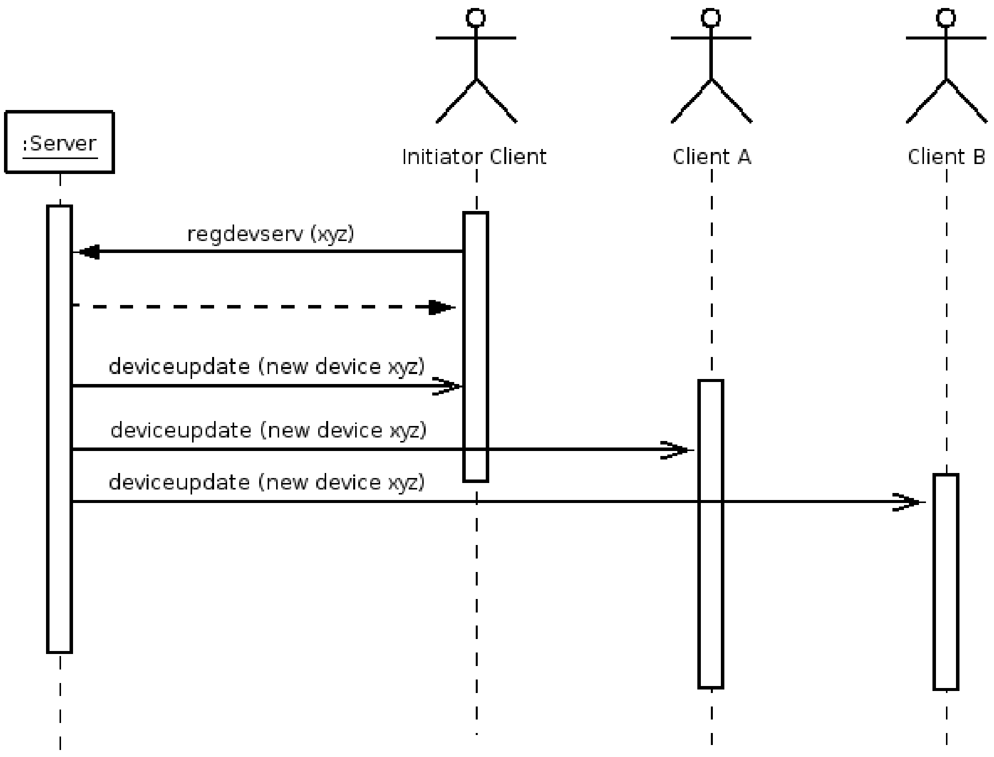

:Server
Initiator Client
Client A
Client B
regdevserv (xyz)
deviceupdate (new device xyz)
deviceupdate (new device xyz)
deviceupdate (new device xyz)

Figure 2.4: device registration

Sometimes consistency must also be provided when altering the state of the server. i.e. When registering a new device or when connecting a new bus, etc. to the ABI system (See figure: 2.4).

2.4 Protocol specification

Clients can consume services which take almost no time to complete. On the other hand the services may take considerable amount of time to complete. So it's important that the API provides both blocking and non-blocking API calls. With a blocking API 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 service. This is a very useful method 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.

Thus a non blocking API will provide the client to use a callback mechanism (or polling mechanism) to get the responses (if any) for a service invocation.

Most of the features presented in the RBC specification ([NB05c]) can basically be regarded as features to be implemented and MUST therefore be provided by this API.

2.5 API abilities and responsibilities

The core principle of this API is to provide a platform for custom client applications and also bundles (See section: 11.3). Hereby the goal that this API strives for is to detach any client relevant concepts (such as an AI or Controlling application) from the underlying communication and reproduction part. Thus increasing abstraction and decreasing low level concerns.

2.6 API Goals

- Independent of any server implementation
- Vice-versa. Server can be changed independently
- Applied Object Oriented Concepts (i.e. Observer notifies about a device change)
- The API must be designed to be portable and expendable to be installed as a separate bundle (See section: 11.3)
- Enhancing development speed, testability and debugging, etc.
- A clear separation of concern (Server and Client)
- Must be capable to provide a stable platform for custom client applications. Concretely speaking the API is destined for logging, intelligence and administrative application programs.
- Comprehensibility
- Easy to use (close to domain)
- Well documented (with accompanying examples)

Part III

Architecture, Design and Implementation

Chapter 3

Architecture and Design

3.1 Overview

This section describes the environment and the core concepts of the RBC API and the role it plays in applications. According to figure: 3.1, the architecture is composed into three main parts. Further the bottom is constructed out of (two) sub-parts.

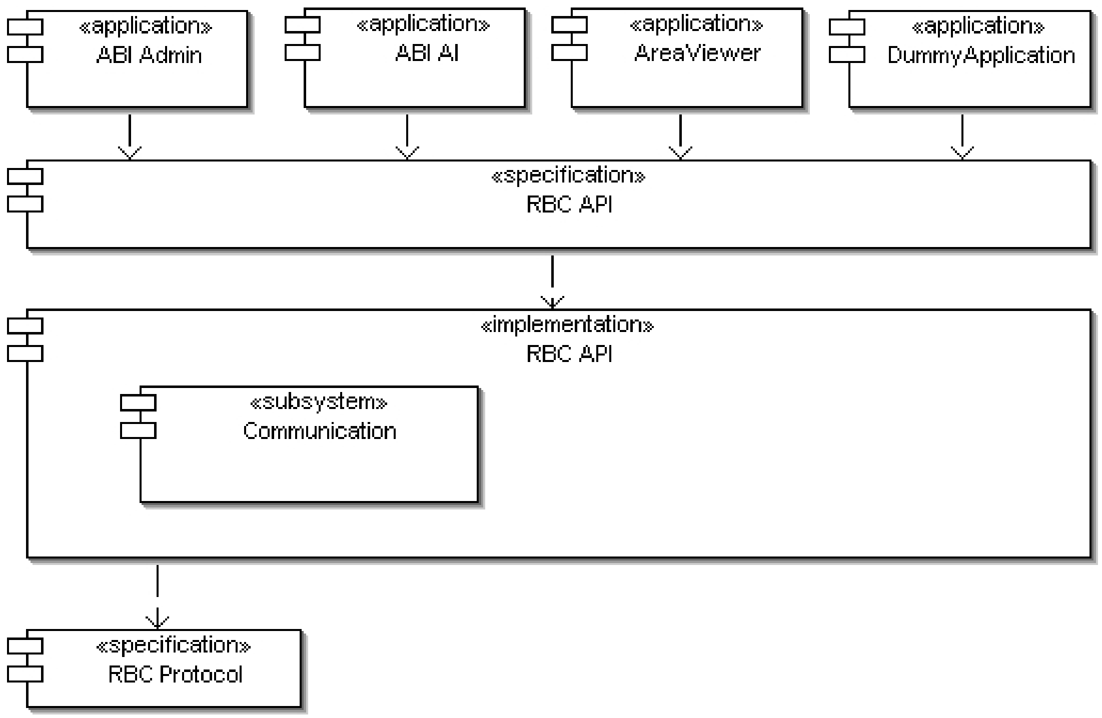


Figure 3.1: Architectural overview

- On top of the architecture we place custom client applications that make use of the actual API. Evidentially one can observe that client applications do not depend on other parts but the RBC specification.
- Underneath we introduce an API specification that is independent of any concrete realizations.
- Having once defined the RBC specification, a set of concrete RBC implementation can be implemented.

The concrete implementation is tightly coupled with a communication subsystem that invokes prescribed protocol specific implementation constraints.

3.2 RBC API Specification

The RBC API Specification defines following abstract classes and interfaces (See figure: 3.2 and sections: 3.2.2 and 3.2.3).

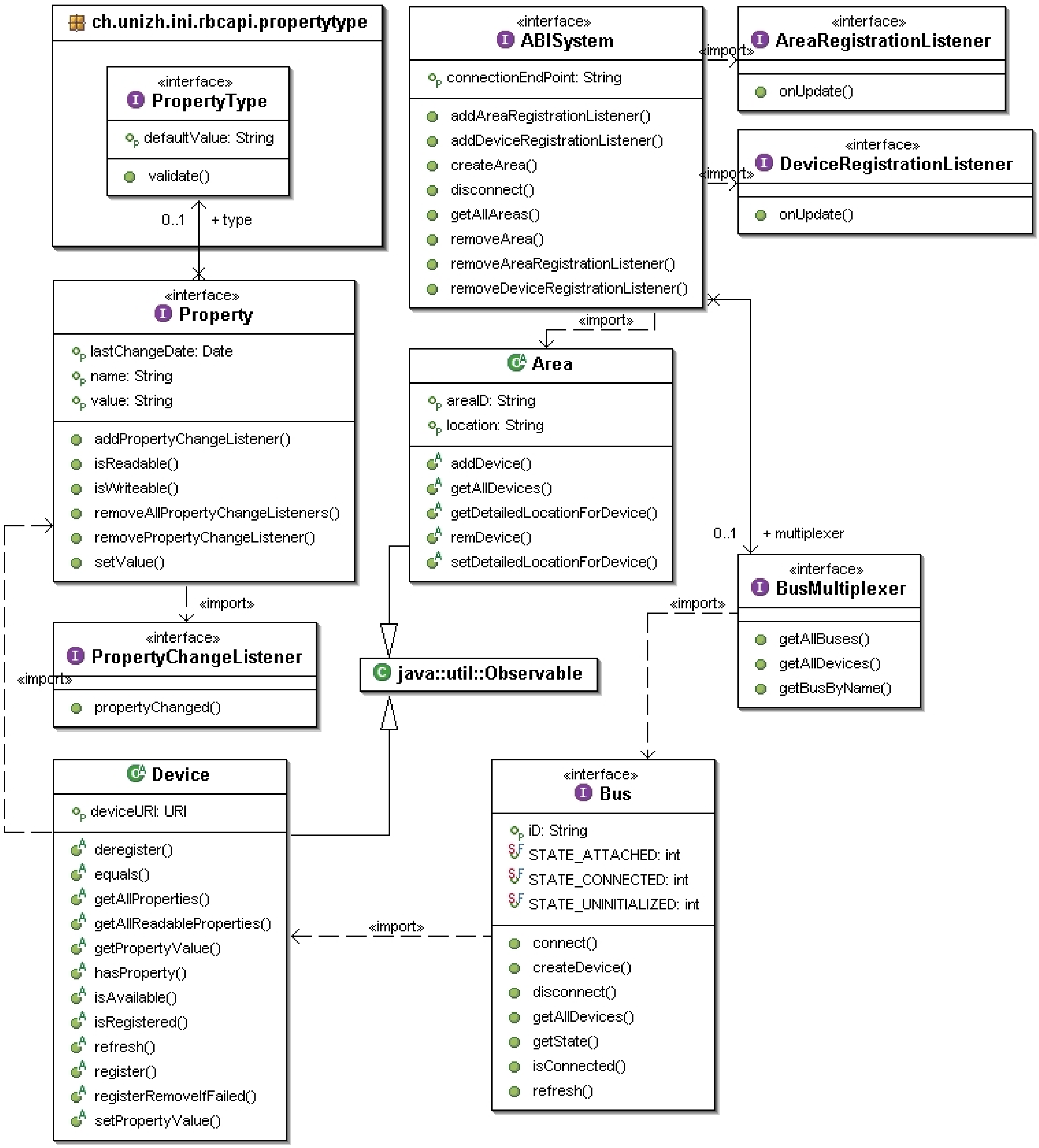


Figure 3.2: API Specification overview

3.2.1 General

The abstract classes and interfaces may depend on external packages provided by the Java Runtime Environment (JRE), which are not listed here. For details on the Java packages, see the Java API documentation. ([JAVa])
Please note that more information can be acquired by the appropriate JavaDoc section ([JAVb]).

3.2.2 Abstract Classes

Quite often API specifications are commonly given as abstract classes and interfaces. Naturally they are composed in a way that they restrict any misleading access and are therefore declared as *internal* as a precaution.
This specification defines two abstract classes each of which needed to be defined as `abstract` since both require to implement the `Observable class` (See sections: 3.2.2.1 and 3.2.2.2).

3.2.2.1 Area

An Area is basically a collection of devices. No other functionalities have been currently defined yet. It should be the abstract representation of a logical structure. For instance it could be a room or something like that.
Additionally each device within an area has an exact detailed location. i.e. window or corridor.
According to section 3.2.3.2 the `abstract Area class` is derived by the `Observable class`. This is necessary since areas can be altered by all clients simultaneously. So it notifies any changing states to appropriate observers.

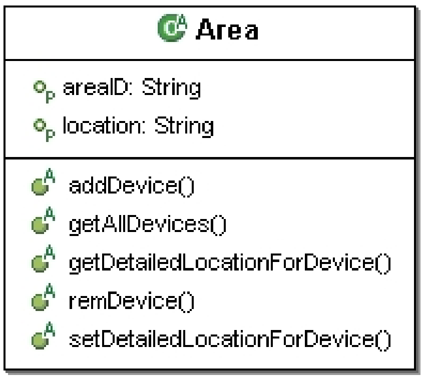


Figure 3.3: Area representation

3.2.2.2 Device

In contrast to the `Bus interface` (See section: 3.2.3.4) which only defines devices, this abstract class is responsible of providing methods `register(), deregister()` which intends to conveys the data to the ABI System for accountancy. So when performing these methods the device must first be created across the `Bus Interface, createDevice()` method.
Please note that if the registration process cannot be completed out of some any reasons, the device provides an additional method which allows the device to be removed completely (`registerRemoveIfFailed()`). This might be useful for certain type of application programs (i.e commonly applied when trying to access a device which does either not exist or not respond)
Other methods such as: `getAllProperties(), getAllReadableProperties(), getPropertyValue()` are quite useful for many applications when acquiring property data.
More information about the property concept can be obtained in our term-project documentation ([NB05b]).

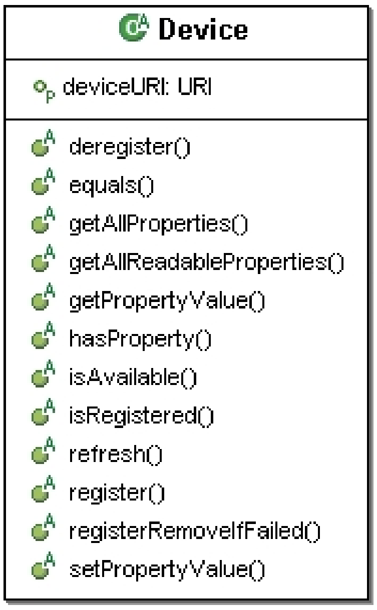


Figure 3.4: Device representation

3.2.3 Interfaces

The interfaces described here usually correspond to either `Listeners`, generic data type or other domain related definitions.

3.2.3.1 Property

Every device (physical or virtual) has a set of properties. A property is actually just a tag value pair as known by the Java API itself. 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 the client-software if additional devices are plugged at runtime.
In order to receive changing property information a method `addPropertyChangeListener()` has been added to this interface that can be registered through any class given by a client application.
For more information about the property concept please consult our term-project documentation ([NB05b]).

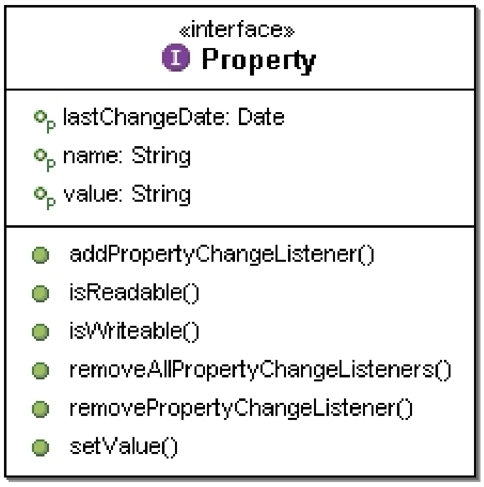


Figure 3.5: Property representation

3.2.3.2 ABISystem

This interface defines the entire ABI system for either a remote or a local access. Practically all calls must be initiated across this interface.
Mainly it provides the capability to `add/remove` listeners which are used by client applications which want to be notified about any added devices or areas. Vice versa this listeners will also be notified when areas and devices have been removed from the system.
To acquire changing state information of existing devices or areas, consult sections 3.2.2.2 and Architecture and Design.RBC API Specification.Abstract Classes.Area instead.

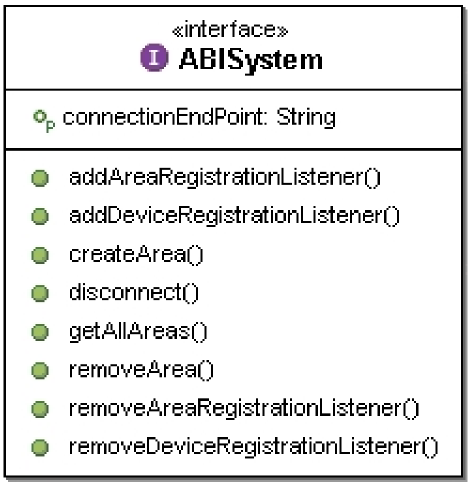

«interface»
ABISystem
connectionEndPoint: String
addAreaRegistrationListener()
addDeviceRegistrationListener()
createArea()
disconnect()
getAllAreas()
removeArea()
removeAreaRegistrationListener()
removeDeviceRegistrationListener()

Figure 3.6: ABISystem representation

3.2.3.3 BusMultiplexer

Just like the ABI System this API provides a small interface that is responsible to multiplex buses.
This interface is commonly implemented by concrete API's when applications need to find all available buses within the ABI System. This happens through the call of `getAllBuses()`

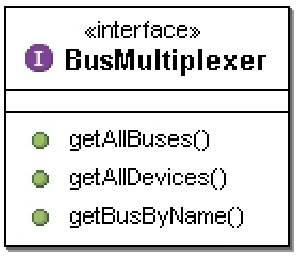


Figure 3.7: BusMultiplexer representation

3.2.3.4 Bus

A client side representation of a bus (i.e. The LonBus or the FalconBus, etc.).
The `Bus interface` provides a set of administrative methods that have been disposed to dealing with buses. It's important to note that this isn't the only interface where new devices can be created by invoking: `createDevice() method`.
The `connect()` method is responsible to hook a given bus to the system. Generally this method is called through the `abstract Device class` when acquiring any specific device information where a connection to the bus is necessary.
Therefore instead of calling `connect()` manually, a direct call across the `abstract Device class` might be better choice.
Because of the fact that every device (including virtual devices) are necessary to be attached by exactly one bus, it's crucial to know that all devices need to be removed from the system upon calling the `disconnect()` method.

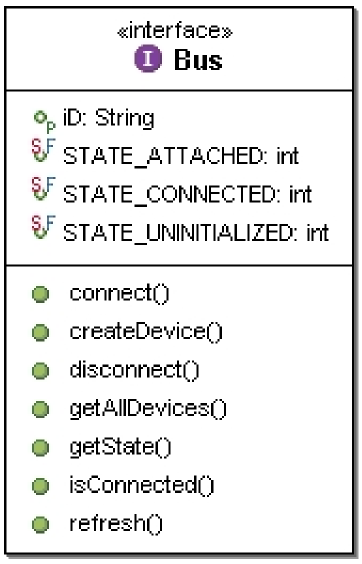


Figure 3.8: Bus representation

3.2.3.5 PropertyType

This interface defines an interface that all "custom" data types need to implement. Currently we provide six different data types. The value of the field type should match one of the data types listed in the first column of table 3.1.

The protocol specification suggests to implement an API that provides an interface that defines all "custom" data types (See table: 3.1) according to the RBC specification. All concrete data types would hereby need

Data type	**Constraint**
EnumType	list of enums: i.e. enum:ON enum:ON
IntegerType	{min, max} pair. i.e. min:0 max:128
FloatType	{min, max} pair. i.e. min:0 max:128
DoubleType	{min, max} pair. i.e. min:0 max:128
BooleanType	NONE
StringType	Given as a regular expression(regex) i.e. we can define a date format:([0-9]+)/([0-9]+)/([0-9]4)

Table 3.1: Data types and their constraints

to implement to this interface. We suggest naming the data types according to the names specified in table 3.1. Further we suggest of providing a validation scheme in form of a method that checks corresponding values (`validate()` method). Note that this is necessary since every data type has i.e. a Min or Max value or if we go further let's say an EnumType would need to check if the value string is equal to one of those enumerated values. You can compare the entire property concept including the property type 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 device provides. i.e. A light might be able to be switched off and on. So we provide the custom tags: writeable, readable and a set of predefined types (IDL) to define a property. With that simple locomotive a client application can easily visualize the properties without having to know about the concrete server in the first place. i.e. A sample GUI application can be accomplished like that:

- A light switch can be visualized, since it's writeable and can therefore be illustrated using a combobox since it's an EnumType or BooleanType.

- A presence sensor might be visualized using a label since its not writeable but readable.

Want more information? Consult the term-project documentation ([NB05b]).

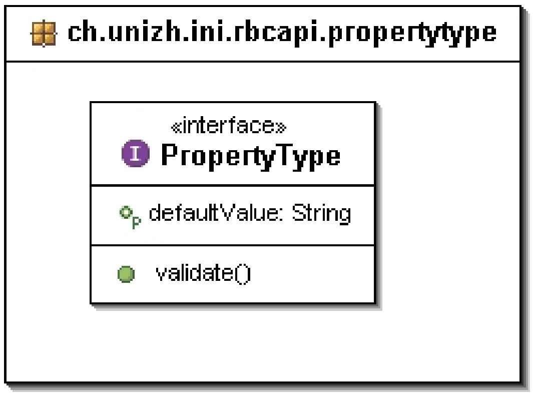


Figure 3.9: PropertyType representation

3.2.3.6 PropertyChangeListener

According to the `Property interface` (See section: 3.2.3.1) any device can register multiple property listeners in order to receive any property changes. i.e. A lightstatus property may provide two different propertyvalues (ON or OFF) that is being sent by the server (considering the Remote implementation of the RBC API). Basically its a callback interface that is used by any class that wants to be notified about any changes of a property for which it has been registered for. So any client that wants to be notified about the property-change must implement the `PropertyChangeListener interface, propertyChanged() method`.

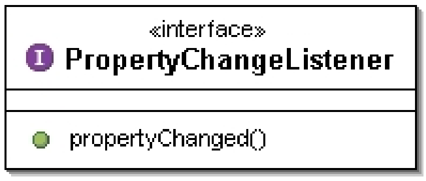


Figure 3.10: PropertyChangeListener representation

3.2.3.7 DeviceRegistrationListener

According to the `ABISystem interface` (See section: 3.2.3.2) any device can be registered or deregistered to/from the ABI System. In order to receive such kind of changes. i.e. When a device has been registered through the (`abstract Device class, registerDevice()`) by a client. Other clients which want to be notified about the registration process must implement the `DeviceRegistrationListener interface, onUpdate() method`.

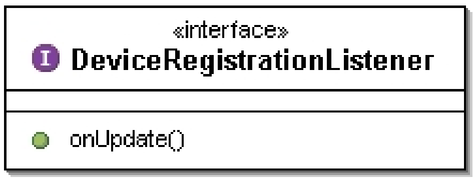


Figure 3.11: DeviceRegistrationListener representation

3.2.3.8 AreaRegistrationListener

According to the `ABISystem interface` (See section: 3.2.3.2) any area can be created or removed to/from the ABI System. In order to receive such kind of changes. i.e. When an area has been created through the (`ABISystem interface, createArea()`) by a client. Other clients which want to be notified about the registration process must implement the `AreaRegistrationListener interface, onUpdate() method`.

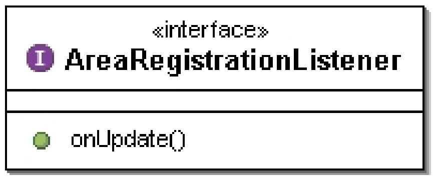


Figure 3.12: AreaRegistrationListener representation

3.2.4 Exceptions

Exceptions (See table: 3.2) usually provide good detailed information about their reason or cause. So when developing GUI applications it is advisable to use something like that:

```java
try
{
    // Do some call that might throw an exception
    bus.connect();
}
catch (Exception e)
{
    // Display an error message on the screen
    JOptionPane.showMessageDialog(null,
        e.getMessage(),
        "Error",
        JOptionPane.ERROR_MESSAGE);
}
```

Its been tried not to use cryptic error messages on the server. So any GUI application may take advantage of this additional option.

Exception types	Description
AreaCreateException	The desired area couldn't be created. This exception usually appears and is being thrown when trying to create an area that does already exist.
AreaRemoveException	The desired area couldn't be removed. This exception usually appears and is being thrown when trying to remove an area that has not existed before.
ConnectException	This exception can be thrown in two different variations. One form of the `ConnectException` is when the client fails to connect to the server, or the other is when the server fails to connect a desired bus. (i.e. falcon bus)
DeviceDeregistrationException	The `DeregisterException` usually appears when the client tries to deregister a device which doesn't even exist in the ABI System.
DeviceRegistrationException	The `DeviceRegistrationException` usually appears when the client tries to register a device which has already been registered or when the busid is unknow or the bus does not support such a deviceurl.
DeviceAlreadyInRoomException	The `DeviceAlreadyInRoomException` appears when a client tries to add a device to an area that has already accounted this device in the first place.
DeviceNotFoundException	The `DeviceNotFoundException` appears when a device cannot be detected from the requested area.
DeviceReadException	The `DeviceReadException` appears upon failing to read the properties of a device.
DisconnectException	The `DisconnectException` appears upon failing to disconnect from a given bus.
InvalidValueException	The `InvalidValueException` appears when the property value of the corresponding property type definition has been violated.
LocationChangeException	The `LocationChangeException` appears when the the device or the area does simply not exist.
PropertyNotExistException	The `PropertyNotExistException` appears when trying to acquire a property by a given name which does not exist in this device.
PropertyNotReadableException	The `PropertyNotReadableException` appears when trying read a property which is not readable.
PropertyNotWriteableException	The `PropertyNotWriteableException` appears when trying write to a property which is not writeable i.e. A presence sensor usually provides only readable properties.

Table 3.2: Exceptions

Chapter 4

Implementation

4.1 Introduction

Since we only want to point out the essentials we keep this chapter short because most of the things have already been covered by the previous chapters. As already stated in the abstract we implemented two different variations of the RBC API specification.
These are: `RBCRemoteImpl.jar` and the `RBCDummyImpl.jar`. The `RBCRemoteImpl.jar` refers to the remote implementation of the API that must be included when doing any real interactions with the RBC Server.
The `RBCDummyImpl.jar` has been implemented with the goal of performing any development activities completely independent of the RBC Server and therefore doesn't need any network connection to be established in the first place. The intention is obviously: During the development phase you might consider to include the `RBCDummyImpl.jar` instead and swap the underlying implementation as soon as the client application has been completed and tested. Thus facilitating the development practices because the `RBCDummyImpl.jar` as the name implies provides a dummy implementation of the ABI System. Concretely speaking it provides a dummy bus, creates some dummy areas and devices and everything (as mentioned) in form of a simulation since nothing really does happen.

The following deployment diagram should mirror a full overview of the current situation (See figure: 4.1):

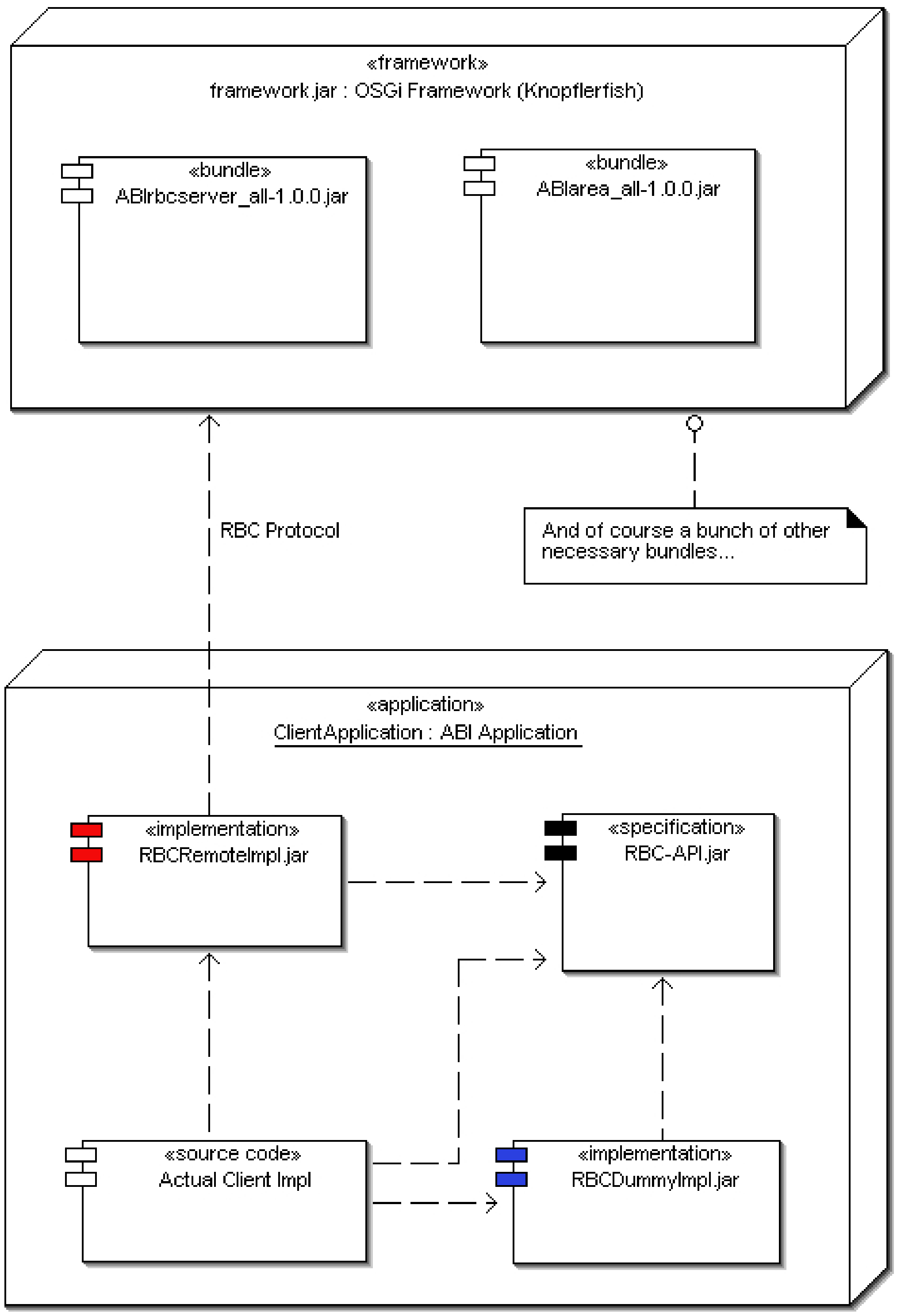


Figure 4.1: The RBC API deployment overview

4.2 Remote RBC API implementation (RBCRemoteImpl)

4.2.1 Overview

The RBCRemoteImpl implements the remote variation of the RBC specification. Since this implementation needs to communicate to the RBC Server ([NB05b]) a communication subsystem has been used (See section: 4.2.2).

4.2.2 Communication Subsystem

The communication subsystem has been 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 is responsible for transferring control and information messages between the client and the server. The communication is done via messages that are exchanged between the client and the server.
For more information about the applied protocol and details on the messages and their format please consult the RBC protocol specification ([NB05c]).

The communication subsystem has been more or less adapted from the subsystem implemented in the ABI RBC Server. So for more details about this architecture consult our term-project documentation ([NB05b]). Nevertheless when looking closely one will notice that both subsystems have been implemented in two different variations. The concrete remote RBC API implementation uses an active open connection in order to establish a connection to the server which in contrast is rather in passive mode (See figure: 4.2).

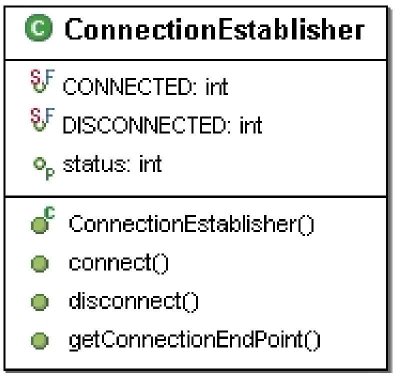


Figure 4.2: ConnectionEstablisher representation

Further distinguishable marks are that the server dispatcher (`MessageDispatcher`) which is launched as a separate (`Thread (Runnable)`) will enqueue any handler tasks to a separate `Thread` in the `ThreadPool class`, 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 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 corresponding handler.
The concrete remote RBC API implementation does not uses this principle since no concurrent usage of the dispatcher is usually necessary.

4.2.3 Code inspection

It wouldn't make sense to cover all the details and classes again since the RBC API Specification already covered the features that must be implemented by each concrete API. But some insights about the Remote

Implementation are now given which might give some hints how it has been realized.

4.2.3.1 ABISystemImpl and InternalABISystem

The ABISystemImpl class (See figure: 4.3) represents the whole ABI-System for remote access. Its signature corresponds to the `ABISystem interface` (See section: 3.2.3.2).
All classes will be created from this class since it serves as the main entry point to any concrete client applications that make use of the remote RBC API. This happens to be the only public class with a public constructor anyway.

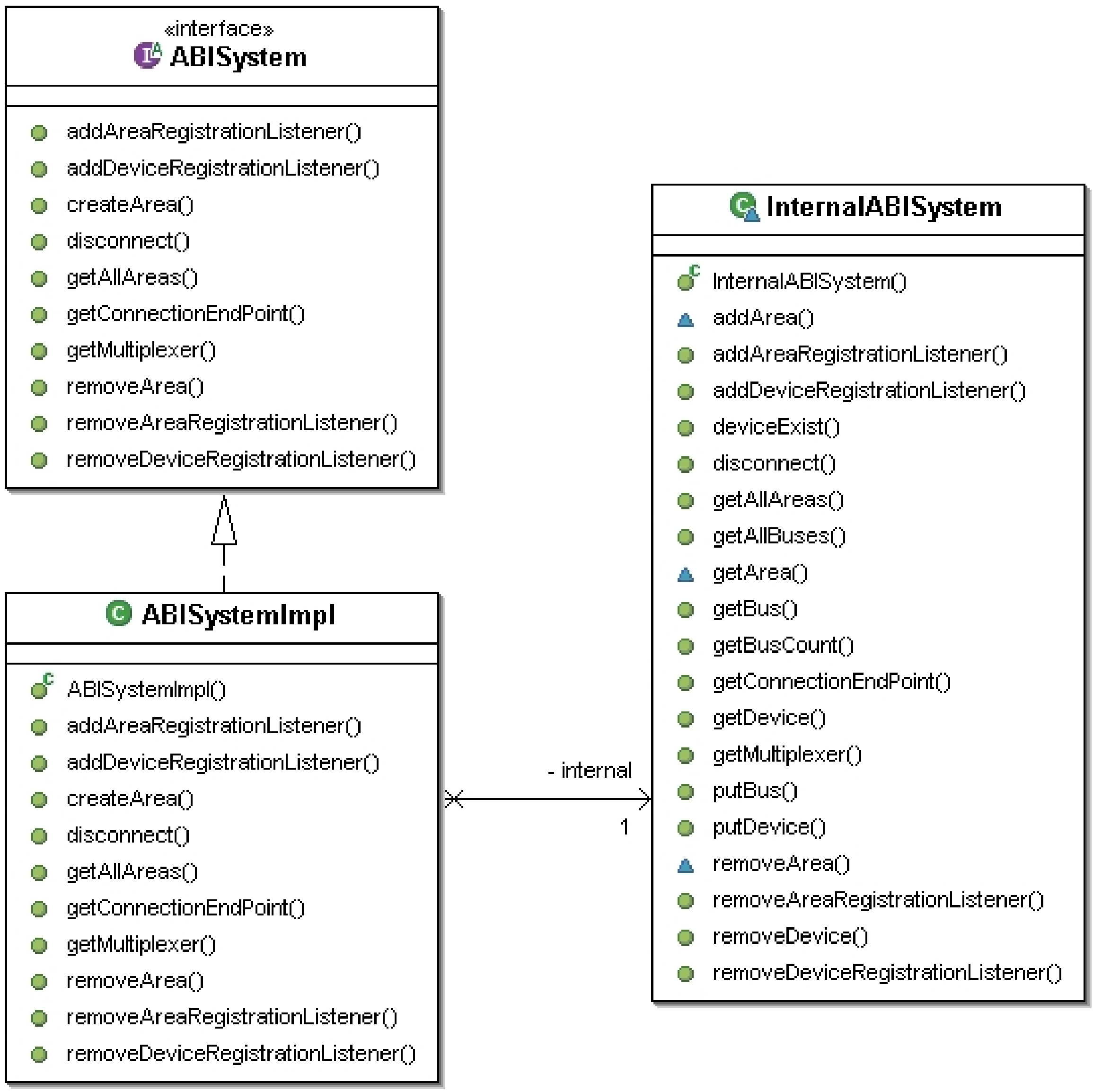


Figure 4.3: ABISystemImpl and the InternalABISystem class

The `InternalABISystem class` (See figure: 4.3) is basically a helper class that has been aggregated by the `ABISystemImpl class`. Its main task is to figure out the side of the event. Specifically speaking an application might have different access rights then the handlers (See section: 4.2.2). This is because on one side the application is allowed to access the API (quite often with restriction) and on the other side the handlers also (quite often with more rights). So some class needed to be sandwiched that solves the problem of indirection in a way that it doesn't affect the visibility of either party (handler and the application side). This might sound very complicated but needed to be done for differ between the two cases.

Further task are to register all necessary communication handlers that correspondingly process incoming messages initiated by the RBC Server ([NB05b]) (Line 4-11).

```
1  public InternalABISystem(String host, int port) throws ConnectException
2  {
3      dispatcher = new MessageDispatcher();
4      dispatcher.registerHandler(ABIPConstants.MSG_PROPERTIESUPDATE,
5              new PropertiesUpdateHandler(this));
6      dispatcher.registerHandler(ABIPConstants.MSG_BUSUPDATE,
7              new BusUpdateHandler(this));
8      dispatcher.registerHandler(ABIPConstants.MSG_DEVICEUPDATE,
9              new DeviceUpdateHandler(this));
10     dispatcher.registerHandler(ABIPConstants.MSG_AREAUPDATE,
11             new AreaUpdateHandler(this));
12     conex = new ConnectionEstablisher(dispatcher);
13
14     try
15     {
16         conex.connect(host, port);
17     }catch (Exception e)
18     {
19         throw new ConnectException(e);
20     }
22     //...
22 }
```

4.3 Dummy RBC API implementation (RBCDummyImpl)

In order to allow a server-independent application development a dummy implementation of the RBC API has been made. Broadly speaking it uses the same components that you should already be familiar with in a simplified way.
More or less it provides the same functionality as the RBCRemoteImpl with the distinguishable difference of that it doesn't communicate with the RBCServer. It accomplishes this by providing a set of dummy devices, areas and buses. (See section: 4.3.2) for a more detailed illustration.

4.3.1 Overview

When having a look at the class diagram one will notice how the different components collaborate with each other. Please note that a bunch of other dependencies such as the listeners have been skipped (See figure: 4.4).

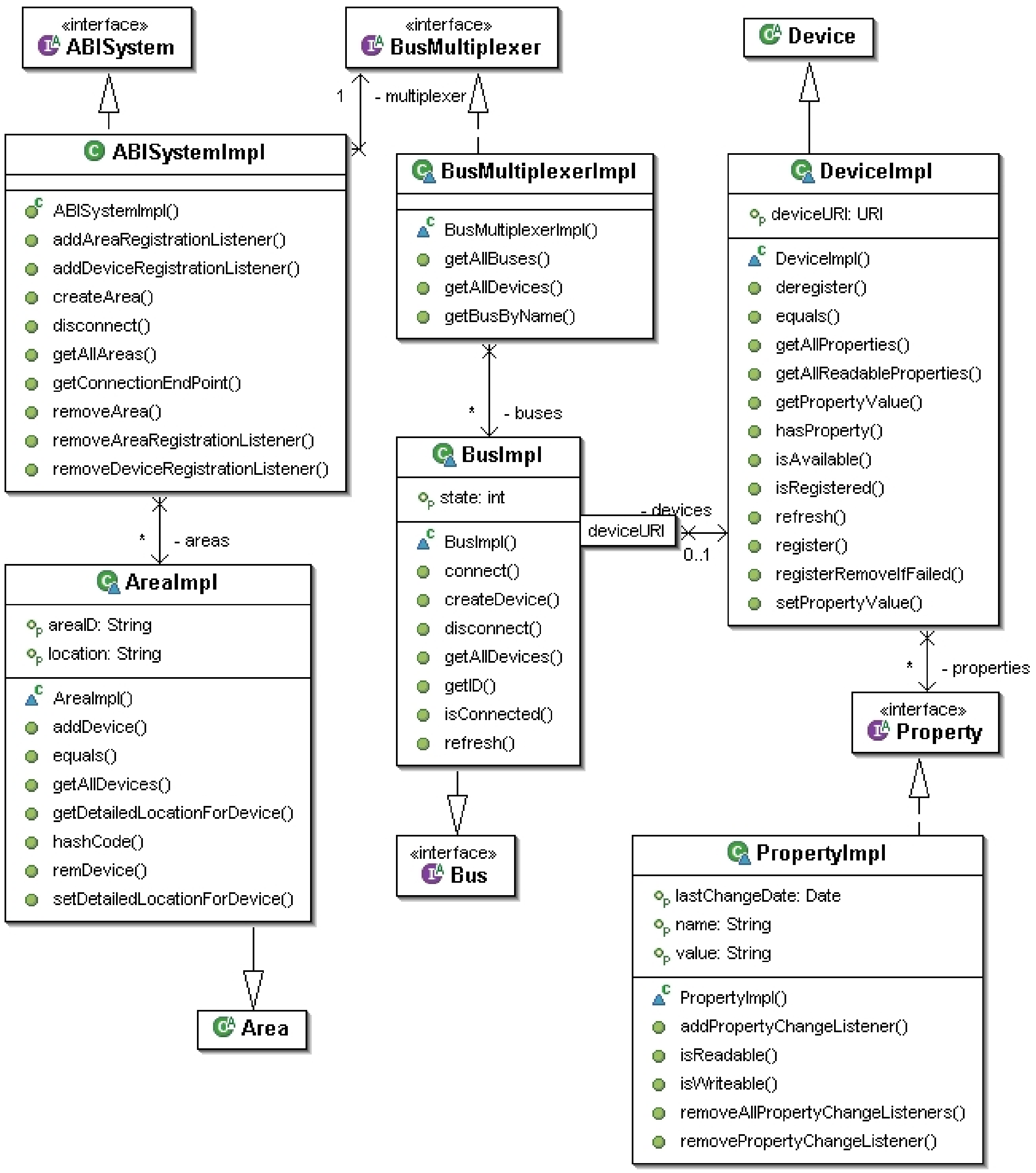


Figure 4.4: RBCDummyImpl overview

4.3.2 Code inspection

When having a close lock at each implementation one will notice that each class instantiates a set of default so called dummy objects.

For instance the `AreaImpl class` creates a set of default dummy devices.

```
devices.add(new DeviceLocationPair("location1",
  new DeviceImpl(new URI("http://1.1.1.1:1/dummy/noservice"))));
```

```
devices.add(new DeviceLocationPair("location2",
  new DeviceImpl(new URI("http://1.1.1.1:2/dummy/noservice"))));
devices.add(new DeviceLocationPair("location3",
  new DeviceImpl(new URI("http://1.1.1.1:3/dummy/noservice"))));
```

The `BusImpl class` creates a local bus facade for dummy Buses. Hereby *2.2.2.2* refers to dummy buses instead of real buses provided by the ABI System. As you can see the Bus is also allowed to create new devices.

```
devID = "http://2.2.2.2:1/"+busID+"/noservice";
devices.put(devID, new DeviceImpl(new URI(devID)));
devID = "http://2.2.2.2:2/"+busID+"/noservice";
devices.put(devID, new DeviceImpl(new URI(devID)));
devID = "http://2.2.2.2:3/"+busID+"/noservice";
devices.put(devID, new DeviceImpl(new URI(devID)));
```

The `DeviceImpl class` creates a set of properties each of which represents a different type to enable real test cases.

```
properties = new Property[6];
properties[0] = new PropertyImpl(
  "boolean","true",true,true,new BooleanType());
properties[1] = new PropertyImpl(
  "enum","A",true,true,new EnumType(new String[]{"A","B","C","D","E"}));
properties[2] = new PropertyImpl(
  "float","1.2",true,true,new FloatType(0,5));
properties[3] = new PropertyImpl(
  "double","1.3",true,true,new DoubleType(0,5));
properties[4] = new PropertyImpl(
  "int","4",true,true,new IntegerType(0,5));
properties[5] = new PropertyImpl(
  "string","asssd",true,true,new StringType(".*d"));
```

4.4 General aspects and principles

4.4.1 Lazy Acquisition Principle

The Lazy Acquisition principle applied in this context defers resource acquisitions to the latest possible time during system execution in order to optimize resource use.
Specifically speaking instead of requesting all devices and their properties we split the task into two parts. Namely we distinguish between a device and its properties in the sense that we don't load the accompanying properties upon calling `Bus, getAllDevices()` which actually returns an array of devicesurls, each of which basically represents a device proxy ([GHJV94]) object (See section: 4.4.2)
This procedure was necessary since we prefer not to block the communication subsystem with long lasting communication tasks which would frankly speaking happen quite often when eagerly acquiring devices and their properties as a tuple.
So when acquiring all possible devices of any available buses i.e. the Falcon bus, a device proxy object is created for each registered device found in the ABI System. In order to acquire all defined or needed device properties from the desired device, the time consuming method: `getAllProperties()` needs to be called from the `Device` (proxy object).
Assuming you have a GUI application that shows all available devices that are currently member of a room (area). When trying to access one of the devices which hasn't been loaded before, the system will acquire all device properties currently known to that device automatically (See section: 4.4.2).
However when using the API you shouldn't bother with this concept since its hidden for any application which make use of this API.

When multiple client are interconnected with the server it might happen that a client receives updates about currently unknown devices, areas or even buses. Such update messages are handled by the current concrete remote RBC API implementation as it creates proxy objects for each new device or area upon identification.

4.4.2 Remote Proxy Principle

The last section briefly introduced the use of proxies. To be more accurate a proxy in our context can be identified as a remote proxy that provides a local representative for an object that resides in a different address space (ABI System, RBC Server).
Generally it provides a stub object similar to the "stub" code in RPC and CORBA. Basically it converts most of the regular object method calls to RBC (Remote Building Control) messages.
The following sequence diagram (See figure: 4.5) shows how a remote proxy is being used when dealing with devices. ([NB05c]).

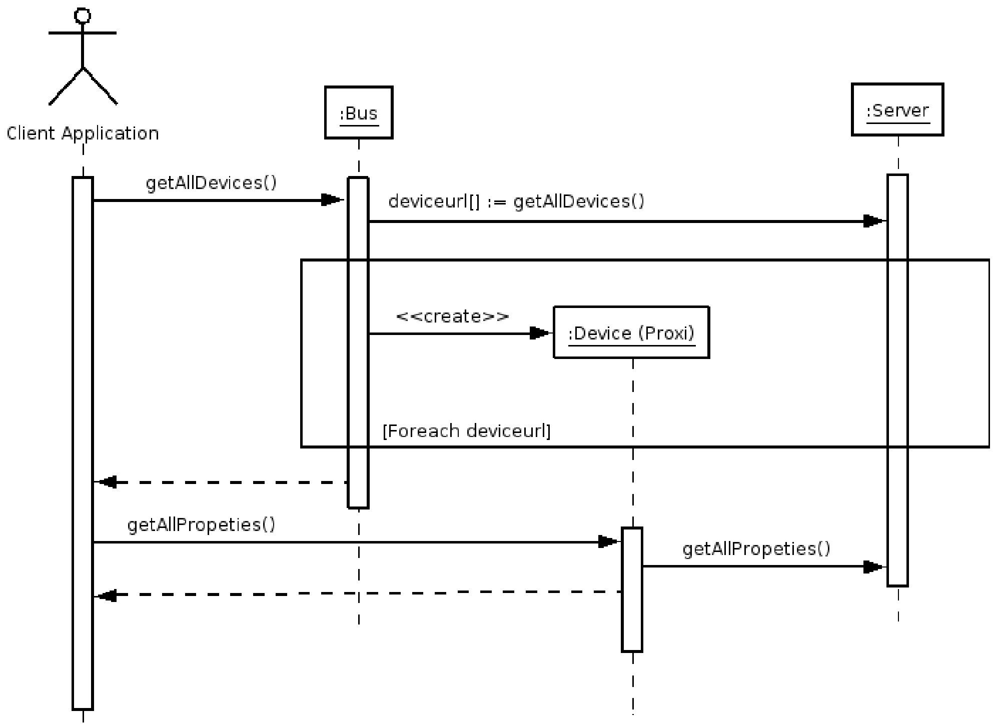


Figure 4.5: Initiated property change

Part IV

RBC API Tutorial

Chapter 5

Introduction

5.1 Introduction

This tutorial introduces you to the Remote Building Control (RBC) programming API. First, the reader will be guided on how to start the Knopflerfish OSGi Framework, and as a following we briefly recap the installation and startup of the ABI System bundles and libraries.
The reason why this might be worth to be rolled up again is that almost each RBC API interaction is basically knocked down to RBC Messages which in turn require a network connection to be established to the RBC Server. So it is fundamental for this tutorial to know how to start the RBC Server that coactive needs to be installed as a separate OSGi bundle within the Framework since it provides some kind of door to the ABI System.
Reasonably we quickly introduce how to setup the RBC API and will then start with the actual tutorial that has been structured into working examples rather then much of theory.
Please note that this tutorial does not intend to cover all the details about this API but should give you enough hints and examples on how to accomplish your own application based on this API.
Furthermore we would like to point out that this tutorial does neither cover any OSGi specifics nor any ABI System details. So for details about any OSGi issues have a look at the Knopflerfish website ([KNO]) or the OSGi Alliance ([OSG]). Also consider our term-project documentation ([NB05b]) that rather covers OSGi related topics.

5.2 Accompanying Executables and Code

All code examples provided by this tutorial and also the accompanying RBC API reference documentation given as JavaDoc can be obtained at: http://www.ini.unizh.ch/∼snufer
Each of the examples (starting from chapter: 7) usually do not contain the entire sources in their full length. Instead this document is meant to provide additional information and help about the examples.
This tutorial is organized as follows:

- First we give some installation instructions on how to start the OSGi Framework.
- As a second we give a short overview on how to install and start the necessary bundles of the ABI System.
- Then we briefly introduce on how to setup the API with an IDE such as the Eclipse Platform ([ECL]).
- The actual tutorial starts by providing a set of the examples each of which is structured like:
  - **Introduction**: A short introduction about the example. What it does, what's its intention, etc.
  - **Code inspection**: We explain some source code snippets used by that example

– **Running the example:** Explains how to start the application and might give some other additional hints.

Chapter 6

Installation

6.1 OSGi Framework installation and startup

6.1.1 Starting the OSGi Framework

We assume that you've already installed the Knopflerfish [KNO] OSGi Framework. Otherwise please consult knopflerfish.org ([KNO]) where you can download the framework accordingly.
Having once installed the framework we recommend to start the graphical user interface provided by knopflerfish. There are three mandatory software packages which are required in order to launch the framework (See figure: 6.1).

- The Java Runtime Environment(JRE) ([JAVa])
- Knopflerfish binary, or source distribution
- and the accompanying ABI System related packages must be available in jars folder of the Knopflerfish installation.

This command initiates the start of the framework with an accompanying graphical user interface.
Open the shell and `cd` to the OSGi directory of the Knopflerfish installation and type:

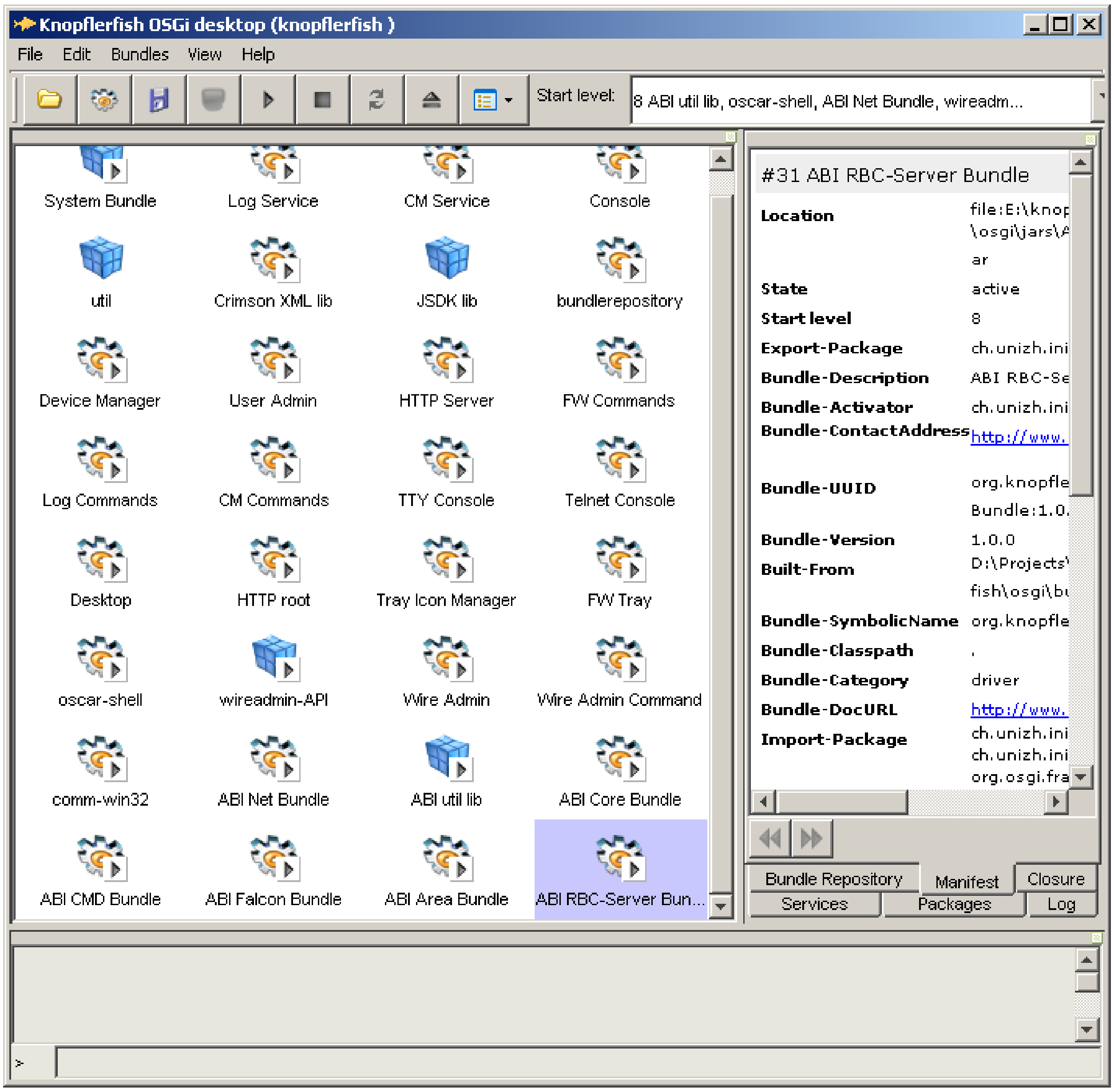
```
java -Xmx128m -jar framework.jar -init
```

Now you have two possibilities on how to integrate the bundles into the framework. You can either install the bundles using the graphical user interface (See section: 6.1.2) or use a telnet console to connect to the framework (See section: 6.1.3).

6.1.2 Installing the bundles with the GUI

Installing the ABI System with the graphical user interface is done by choosing the ABI System bundles with the help of the file dialog. The bundles are then installed and started. The recommended order of installing the bundles is as follows:

1. wireadmin/wireadmin_api-1.0.0.jar
2. wireadmin/wireadmin-1.0.1.jar
3. comm-win32/comm-win32_all-1.0.0.jar



#31 ABI RBC-Server Bundle

Location	file:E:\knop \osgi\jars\A ar
State	active
Start level	8
Export-Package	ch.unizh.ini
Bundle-Description	ABI RBC-Se
Bundle-Activator	ch.unizh.ini
Bundle-ContactAddress	http://www.
Bundle-UUID	org.knopfle Bundle:1.0.
Bundle-Version	1.0.0
Built-From	D:\Projects\ fish\osgi\bu
Bundle-SymbolicName	org.knopfle
Bundle-Classpath	.
Bundle-Category	driver
Bundle-DocURL	http://www.
Import-Package	ch.unizh.ini ch.unizh.ini org.osgi.fra

Figure 6.1: The Knopflerfish OSGi Desktop

4. ABInetmonitor/ABInetmonitor_all-1.0.0.jar (currently in use but will be evicted at a later point of time)
5. ABIutil/ABIutil-1.0.0.jar (currently in use but will be evicted at a later point of time)
6. ABIcoreapi/ABIcoreapi_all-1.0.0.jar
7. ABIFalcon/ABIFalcon-1.0.0.jar
8. ABIArea/ABIArea_all-1.0.0.jar
9. ABIrbcserver/ABIrbcserver_all-1.0.0.jar

6.1.3 Installing the bundles with telnet

You can either telnet to the OSGi framework or you can manually include the bundles by simply typing:

```
telnet localhost
Knopflerfish OSGi console
login:your login-name
password:your password
```

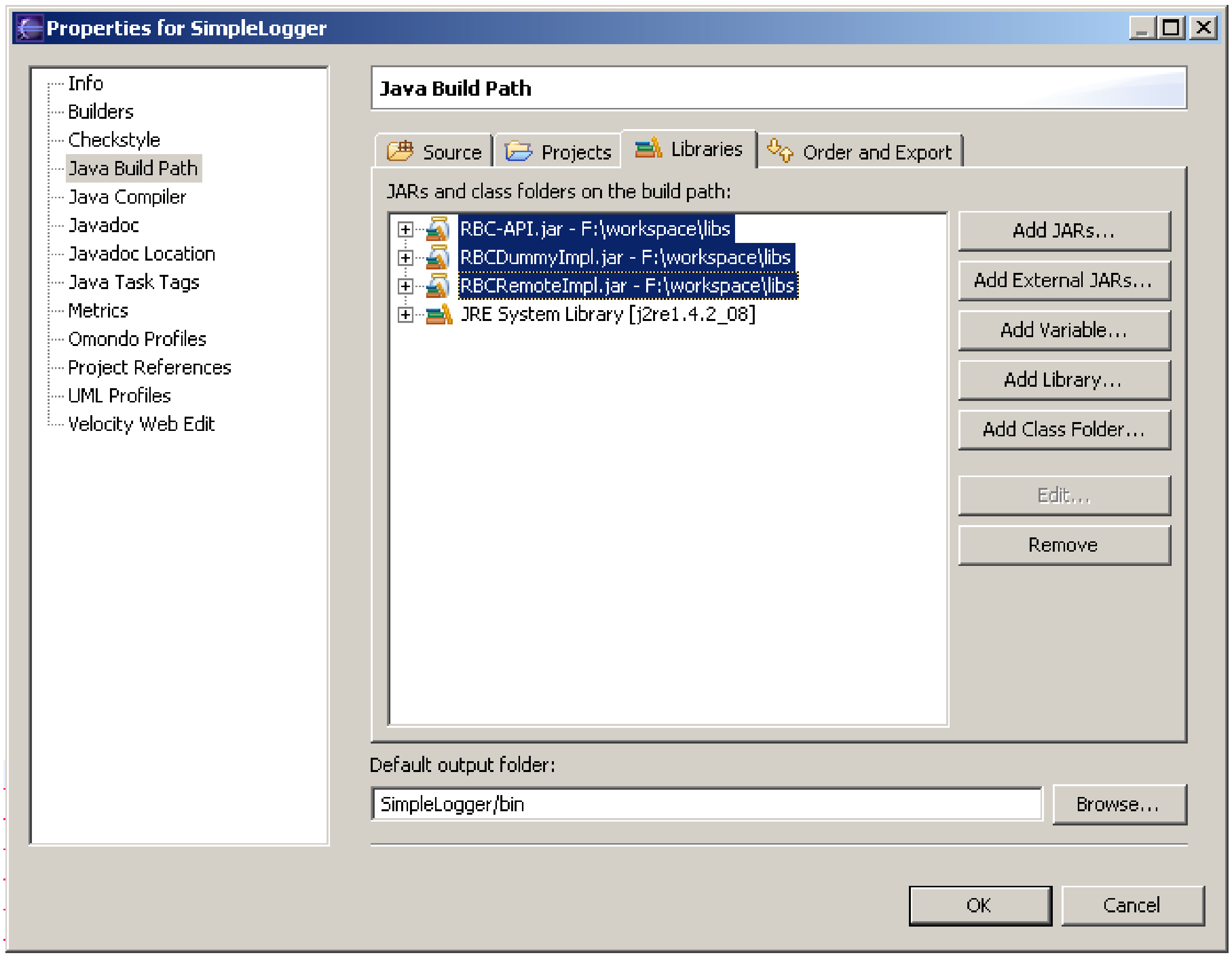
Now that you are connected to the framework's telnet server you are authorized to install the bundles with the following commands (assuming that you've copied all ABI bundles to the frameworks's jar directory):

```
install file:jars/wireadmin/wireadmin_api-1.0.0.jar
        file:jars/wireadmin/wireadmin-1.0.1.jar
        file:jars/comm-win32/comm-win32_all-1.0.0.jar
        file:jars/ABInetmonitor/ABInetmonitor_all-1.0.0.jar
        file:jars/ABIutil/ABIutil-1.0.0.jar
        file:jars/ABIcoreapi/ABIcoreapi_all-1.0.0.jar
        file:jars/ABIFalcon/ABIFalcon-1.0.0.jar
        file:jars/ABIArea/ABIArea_all-1.0.0.jar
        file:jars/ABIrbcserver/ABIrbcserver_all-1.0.0.jar

start "Wire Admin"
      "ABI Net Bundle"
      "ABI Core Bundle"
      "ABI Falcon Bundle"
      "ABI Area Bundle"
      "ABI RBC Server"
```

6.2 RBC API installation

The API is structured into three different `jar` files (See figure: 4.1). For more information about each specific implementation consult chapter 4.

Each example provided below actually explains how to compile and start each of the application. However for the sake of simplicity we recommend to use an IDE such as the Eclipse Platform ([ECL]) in order to simplify any development with the API.
So when creating a new project with Eclipse make sure that you import the `RBC-API.jar` **and** either the `RBCDummyImpl.jar` **or** the the `RBCRemoteImpl.jar` in your Java build path (See figure: 6.2) Otherwise, you will not be able to access the implementation classes and the needed specification provided by all jar files.



Info
Builders
Checkstyle
Java Build Path
Java Compiler
Javadoc
Javadoc Location
Java Task Tags
Metrics
Omondo Profiles
Project References
UML Profiles
Velocity Web Edit

Java Build Path

Source | Projects | Libraries | Order and Export

JARs and class folders on the build path:

RBC-API.jar - F:\workspace\libs
RBCDummyImpl.jar - F:\workspace\libs
RBCRemoteImpl.jar - F:\workspace\libs
JRE System Library [j2re1.4.2_08]

Default output folder:

SimpleLogger/bin

Figure 6.2: Install the RBC API in the eclipse platform

Chapter 7

Creating your first RBC Application

7.1 The SimpleLightGUI Application

7.1.1 Introduction

The goal of this chapter is to explain some basic interactions with the underlying API. For this purpose we developed a simple GUI application that is capable of controlling a Ffalcon light in the sense that it provides a ”light switch” that turns on and off the appropriate light. Please make sure that the Falcon bus and the appropriate Falcon light have been connected to the ABI System that should have already been launched before.
When starting the application you should see the following window (See figure: 7.1):

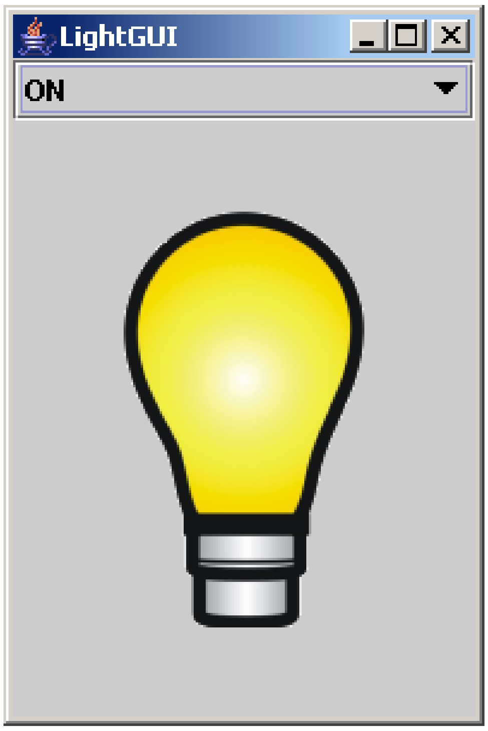


Figure 7.1: SimpleLightGUI application

7.1.2 Code inspection

Again, the entire example can be downloaded from http://www.ini.unizh.ch/~snufer

Here is the code that creates `ABISystem` object (See section: 3.2.3.2) and instantiates the GUI component class `LightGUI`.

```
ABISystem abisystem = null;
try
{
    abisystem = new ABISystemImpl("localhost", 1234);
    new LightGUI(abisystem);
}
catch (ConnectException e)
{
    e.printStackTrace();
}
```

In this example we differed between GUI (`initGUI()`) and ABI (`initABI()`) initialization as far as it was possible. Having a glance back to the documentation of the `ABISystem` class (See section: 3.2.3.2) one will notice that this class is responsible for retrieving necessary bus information across the `BusMultiplexer` (See section: 3.2.3.3) as we can see in line 9.
We then create the device, using the recently created bus object (Line 12).
In line 14-22 we retrieve the desired property name *lightstatus* that has been defined by the concrete device.

```
1  private static final String BUS = "falcon.bus-1.0";
2  private static final String SERVICENAME = "LightService?fqcn=...";
3  // ...
4
5  private void initABI() throws ConnectException, URISyntaxException,
6          DeviceRegistrationException
7  {
8      // Create the bus
9      this.bus = this.abisystem.getMultiplexer().getBusByName(BUS);
10     // Connect the bus to the ABI System
11     this.bus.connect();
12     this.device = this.bus.createDevice(new URI("http://0.0.0.0:35/" + BUS
13             + "/" + SERVICENAME));
14     this.properties = this.device.getAllProperties();
15     for (int i = 0; i < properties.length; i++)
16     {
17         if (properties[i].getName().equals("lightstatus"))
18         {
19             lightstatusProperty = properties[i];
20             break;
21         }
22     }
23 }
```

The following code shows how the necessary GUI components have been initialized. Additional attention should be paid to line 6, 22 and 28.

- **Line 6** extracts all available property values of the lightstatus property in order to initialize the combobox items. Of course this is only possible when having the prior knowledge of the property type. In this case we have: `EnumType` (See table 3.1).

- **Line 22 and 28** reading out the property values which are being changed upon repaint which is among other things initiated by observer updates (See further below).

```
1  private void initGUI() throws IOException, PropertyNotReadableException
2  {
3      this.setSize(200, 300);
4      this.getContentPane().setLayout(new BorderLayout());
5      // Retrieve the property-values of the lightstatus property
6      String comboValues[] = ((EnumType) this.lightstatusProperty.getType())
7              .getEnumValues();
8      this.combobox = new JComboBox(comboValues);
9      // Set combobox
10     this.combobox.setSelectedItem(this.lightstatusProperty.getValue());
11
12     this.getContentPane().add(combobox, BorderLayout.NORTH);
13     this.combobox.addActionListener(new ComboBoxListener());
14     this.panel = new JPanel()
15     {
16
17         public void paint(Graphics g)
18         {
19             super.paint(g);
20             try
21             {
22                 if (lightstatusProperty.getValue().equals("ON"))
23                 {
24                     g.drawImage(lightImageON, (getWidth() - lightImageON
25                             .getWidth()) / 2, (getHeight() - lightImageON
26                             .getHeight()) / 2, null);
27                 }
28                 else if (lightstatusProperty.getValue().equals("OFF"))
29                 {
30                     g.drawImage(lightImageOFF, (getWidth() - lightImageOFF
31                             .getWidth()) / 2, (getHeight() - lightImageOFF
32                             .getHeight()) / 2, null);
33                 }
34             }
35             catch (PropertyNotReadableException e)
36             {
37                 e.printStackTrace();
38             }
39         }
40     };
41
42     this.getContentPane().add(panel, BorderLayout.CENTER);
43     initImages();
44
45     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
46     this.setVisible(true);
47 }
```

Most of the initialization work is hereby completed.
In order to be notified about any device property changes following steps are further necessary:

1. This class must implement an `Observer` in order to receive any updates from the server.

```
public class LightGUI extends JFrame implements Observer
```

```java
{
   //...
}
```

2. Implement the `update()` method that is required when implementing the `Observer interface`. The `SwingUtilities.invokeLater(new Runnable()` enhances any drawings that must be performed upon each update. Specifically speaking it causes the `run()` method to be executed asynchronously on the AWT event dispatching thread.

```java
public void update(Observable o, Object arg)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            try
            {
                // Set combobox
                 combobox.setSelectedItem(lightstatusProperty.getValue());
                 // Repaint light-bulb
                 repaint();
             }
             catch (PropertyNotReadableException e)
             {
                 e.printStackTrace();
             }
         }
     });
 }
```

3. Register ourselves as Observer (Line 9). This can easily be accomplished in the constructor among other initializations state above.

```java
1  public LightGUI(ABISystem _abisystem)
2  {
3      super("LightGUI");
4     this.abisystem = _abisystem;
5      try
6      {
7          initABI();
8          initGUI();
9          this.device.addObserver(this);
10      }
11      catch (Exception e)
12      {
13          e.printStackTrace();
14      }
15  }
```

Of course one might have missed the code that is reacting when the combobox is being altered. The following code does this:

```java
private class ComboBoxListener implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        String val = (String) combobox.getSelectedItem();
6          try
7          {
8             lightstatusProperty.setValue(val, "JCraft Team");
9          }
10          catch (InvalidValueException e)
11          {
12             e.printStackTrace();
13          }
14          catch (PropertyNotWriteableException e)
15          {
16             e.printStackTrace();
17          }
18     }
19  }
```

As one can observe it is quite easy to change the property value of that device (See line 8).
Nobody will actually notice that a small middleware is involved in all of this.

7.1.3 Running the example

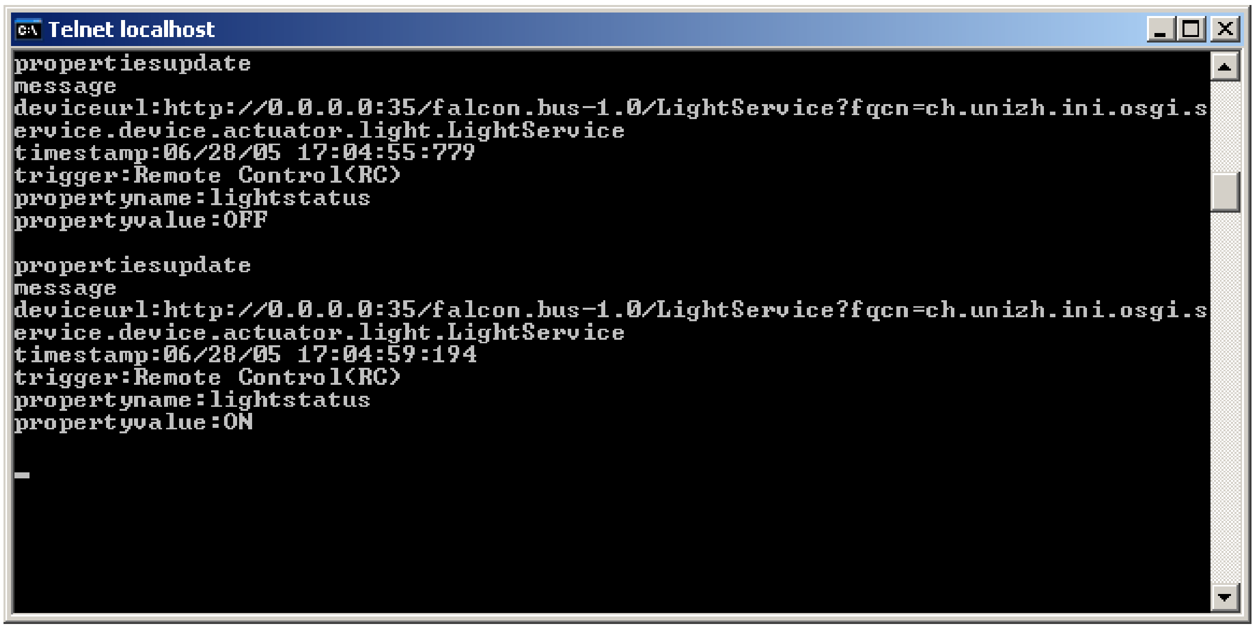
Before running this example, open a shell and telnet to the ABI System as follows:

```
telnet localhost 1234
```

In this case we use *localhost* as our host assuming that the entire ABI System is launched from the same machine as the application. So this could be any valid address where an ABI System might be running.

Having once connected to the server you can start the application and you will be able to switch on and off the lights by simply switching the combobox items.
Now have a brief look to the telnet client.

When switching the combobox you should be receiving RBC Messages such as those illustrated in figure 7.2. Of course this isn't necessary but it demonstrates how the messages are being sent to the ABI RBC Server.



```
propertiesupdate
message
deviceurl:http://0.0.0.0:35/falcon.bus-1.0/LightService?fqcn=ch.unizh.ini.osgi.s
ervice.device.actuator.light.LightService
timestamp:06/28/05 17:04:55:779
trigger:Remote Control(RC)
propertyname:lightstatus
propertyvalue:OFF

propertiesupdate
message
deviceurl:http://0.0.0.0:35/falcon.bus-1.0/LightService?fqcn=ch.unizh.ini.osgi.s
ervice.device.actuator.light.LightService
timestamp:06/28/05 17:04:59:194
trigger:Remote Control(RC)
propertyname:lightstatus
propertyvalue:ON
```

Figure 7.2: SimpleLightGUI: RBC Messages over telnet

Chapter 8

How to use Areas in a RBC Application?

8.1 The SimpleAreas Application

8.1.1 Introduction

The goal of this chapter is to get an insight on how areas are handled by the API. For this purpose we developed a simple console application that performs some area calls. i.e. It allows to create and delete areas and lists all available devices from any desired area. Please make sure that the ABI system is available.

When starting the application you should see a window corresponding to figure 8.2.
Note that any operations such as create or delete areas affect the ABI system in the way that it actually creates those areas in the framework itself. This means when the ABI area bundle ([NB05b]) suddenly stops i.e. upon the stop command or when the entire framework has been shutdown, the ABI system will store the current area context into a XML file in order to be able to boot strap and reinitialize the areas back to their previous state since they actually do not depend on any buses and neither devices.

8.1.2 Code inspection

In order to be able to connect to the ABI System Server (RBC Server) the `ABI System` (See section: 3.2.3.2) needs to be instantiated (Line 5, 21-25).

```
1  public void run()
2  {
3      try
4      {
5          init();
6          mainLoop();
7      }
8      catch(ConnectException e)
9      {
10         e.printStackTrace();
11     }catch (IOException e)
12     {
13         e.printStackTrace();
14     }
15     finally
16      {
17          abiSystem.disconnect();
18      }
19  }
20
21  private void init() throws ConnectException
22  {
23      abiSystem = new ABISystem(SERVER_NAME, SERVER_PORT);
24      inputReader = new BufferedReader(new InputStreamReader(System.in));
25  }
```

After the initialization has been completed the menu loop is being launched and the menu options presented. Upon the matching console inputs (H,L,A,R,D,E) following area actions are possible (Line 8-27).

```
1  private void mainLoop() throws IOException
2  {
3      //...
4
5      // Read input character
6      switch(command.charAt(0))
7      {
8          case 'H':
9              help();
10             break;
11         case 'L':
12             listAllAreas();
13             break;
14         case 'A':
15             addArea();
16             break;
17         case 'R':
18             removeArea();
19             break;
20         case 'D':
21             devicesInArea();
22             break;
23         case 'E':
24             exit();
25             break;
26         default:
27             help();
28
29     }
30     //...
```

API specific constraints are now briefly explained.

Option L, listAllAreas() This is the central code that retrieves all available areas (`listAllAreas()`) (Line 1-8).

```
1  Area[] allAreas = abiSystem.getAllAreas();
2  //...
3
4  for (int i = 0; i < allAreas.length; i++)
5  {
6     String areaID = allAreas[i].getAreaID();
```

```
  String areaLocation = allAreas[i].getLocation();
  //...
}
```

Option A, addArea() When creating a new area simply call `createArea()` on the `ABISystem class`.

```
//Create the area
try
{
  abiSystem.createArea(areaID,areaLocation);
  System.out.println("Area successful created");
}catch (AreaCreateException e)
{
  printError("Area creation failed",e.getMessage());
}
```

Option R, removeArea() In order to remove an area, we must first find it (Line 4-10). When deleting an area simply call `removeArea()` (Line 13) on the `ABISystem class`.

```
1  //Get the right Area
2  Area areaToRemove = null;
3  Area[] allAreas = abiSystem.getAllAreas();
4  for (int i = 0; i < allAreas.length; i++)
5  {
6    if(allAreas[i].getAreaID().equals(areaID))
7    {
8      areaToRemove= allAreas[i];
9
10   }
11 }
12 //..
13 abiSystem.removeArea(areaToRemove);
```

Option D, devicesInArea() Analogical to `removeArea()` the desired area must first be discovered. Then we invoke `areaToShow.getAllDevices()` (Line 2) that retrieves all devices currently located in that area. In order to obtain each specific device location you can make use of `getDetailedLocationForDevice()` (Line 6).

```
1  // areaToShow refers to the desired area
2  Device[] dev = areaToShow.getAllDevices();
3  //...
4  for (int i = 0; i < dev.length; i++)
5  {
6    String detLocation = areaToShow.getDetailedLocationForDevice(dev[i]);
7    String devID = dev[i].getDeviceURI().toString();
8    //...
9  }
```

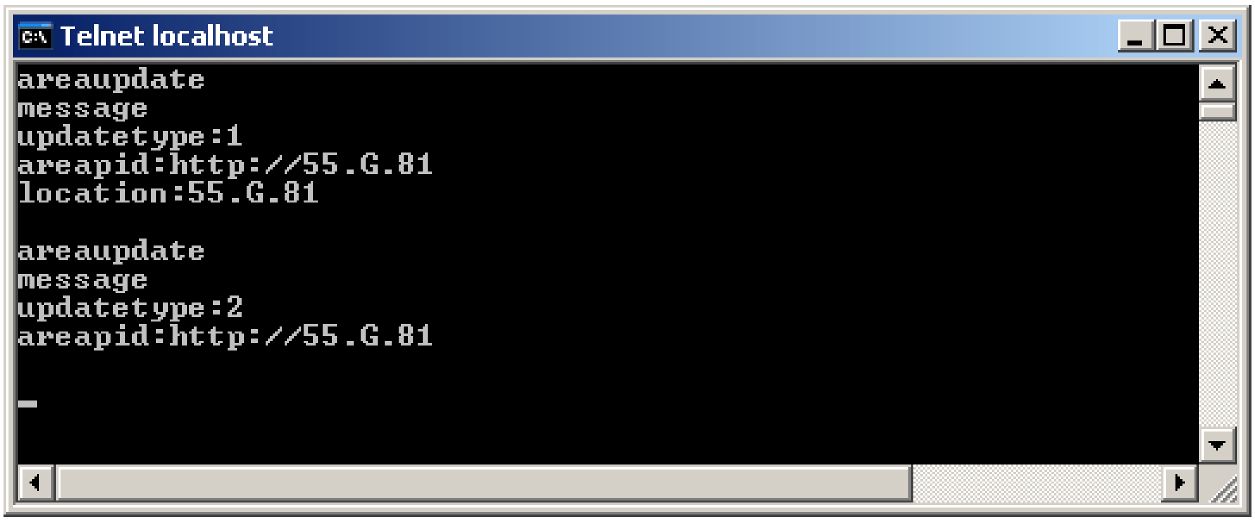
Noticeably, area operations are quite simple and easy to use.

8.1.3 Running the example

Before running this example, you also might want to observe the messages that are being sent between the participants. So you can do this as explained in the previous chapter (See section: 7.1.3).
Note that you only receive update messages so don't expect any others. When you like to observe the entire traffic we suggest packet sniffers such as Ethereal ([ETH]) or Packetyzer ([PAC]).

Having once connected to the server you can start the application and you will be able to create and delete areas, etc. with that program. Again have a brief look to the telnet client (See figure: 8.1).

When creating or removing any areas you should be receiving RBC Messages such as those illustrated in figure 8.1.
Of course this isn't necessary but it demonstrates how the messages are being sent to the ABI RBC Server.



```
areaupdate
message
updatetype:1
areapid:http://55.G.81
location:55.G.81

areaupdate
message
updatetype:2
areapid:http://55.G.81
```

Figure 8.1: SimpleAreas: RBC Messages over telnet

8.1.3.1 Compile and Startup

`cd` into the `src` directory of the SimpleArea application.

- Windows
  -How to compile the application:

  ```
  javac -classpath ../../libs/RBC-API.jar;../../libs/RBCRemoteImpl.jar;. *.java
  ```

  -How to run the application.

  ```
  java -cp ../../libs/RBC-API.jar;../../libs/RBCRemoteImpl.jar;. SimpleAreas
  ```

- Linux
  -How to compile the application:

  ```
  javac -classpath ../../libs/RBC-API.jar:../../libs/RBCRemoteImpl.jar:. *.java
  ```

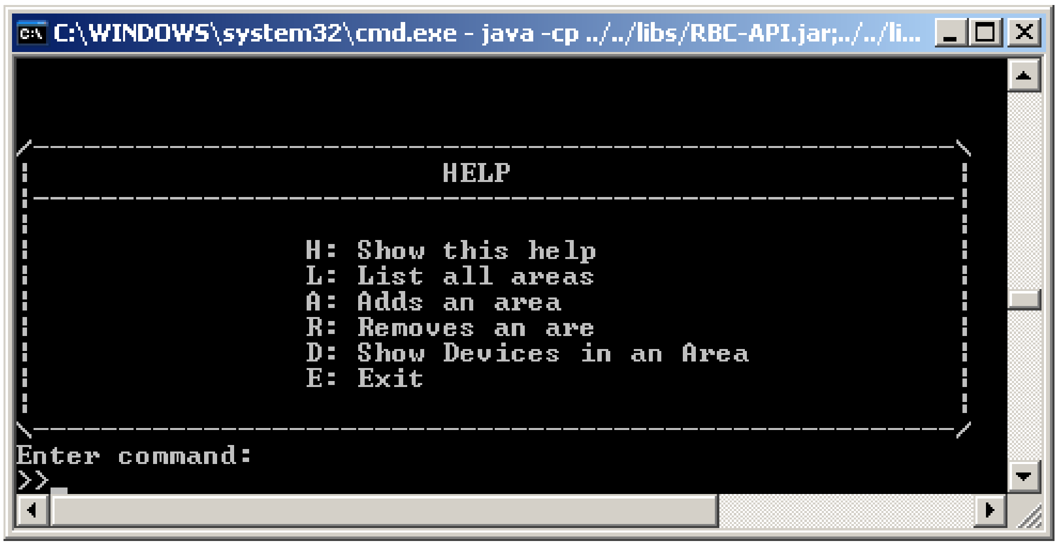
  -How to run the application:

  ```
  java -cp ../../libs/RBC-API.jar:../../libs/RBCRemoteImpl.jar:. SimpleAreas
  ```

- Eclipse Platform
  Include the RBC-API.jar as explained in the beginning of the tutorial (See section: 6.2).

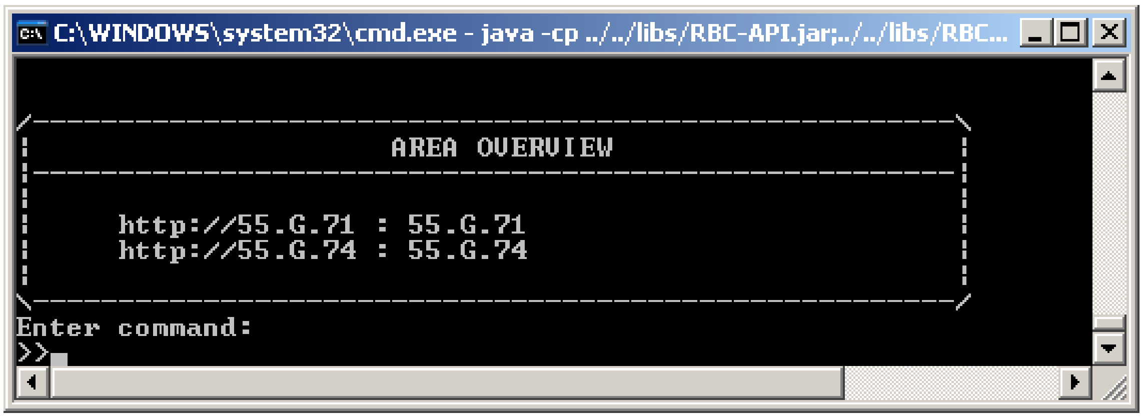
When the compilation and the startup were successful you should see following window (See figure: 8.2).

When pressing **L** you should see all available areas currently known by the ABI System (See figure: 8.3).



```
C:\WINDOWS\system32\cmd.exe - java -cp ../../libs/RBC-API.jar;../../li...

/------------------------------------------------------------\
|                            HELP                            |
|------------------------------------------------------------|
|                                                            |
|                  H: Show this help                         |
|                  L: List all areas                         |
|                  A: Adds an area                           |
|                  R: Removes an are                         |
|                  D: Show Devices in an Area                |
|                  E: Exit                                   |
|                                                            |
\------------------------------------------------------------/
Enter command:
>>
```

Figure 8.2: SimpleAreas Application



```
C:\WINDOWS\system32\cmd.exe - java -cp ../../libs/RBC-API.jar;../../libs/RBC...

/------------------------------------------------------------\
|                        AREA OVERVIEW                       |
|------------------------------------------------------------|
|                                                            |
|      http://55.G.71 : 55.G.71                              |
|      http://55.G.74 : 55.G.74                              |
|                                                            |
\------------------------------------------------------------/
Enter command:
>>
```

Figure 8.3: SimpleAreas overview of all existing areas

Chapter 9

A Simple Logging Service

9.1 SimpleLogger

9.1.1 Introduction

This chapter introduces some features which may be quite useful for other applications to provide. Imagine two client applications that register or deregister devices from the ABI System. For this purpose this API supports listeners that can be registered by any class which wants to receive such information. This example incorporates the listeners in the form of a logging service that records such environmental changes. Other changes which do not effect the environment in form of an enrichment or the opposite are to be implemented as `Observer`. In order to be notified about environmental changes such as Falcon light status or presence status changes this might be meaningful. Since a logging service needs to know about such changes as well, we provide two classes called `MyAreasObserver` and `MyDevicesObserver` that record such alterations.

In order to keep the source code within one file the listeners and the observer classes have been implemented as private classes (See figure: 9.1).

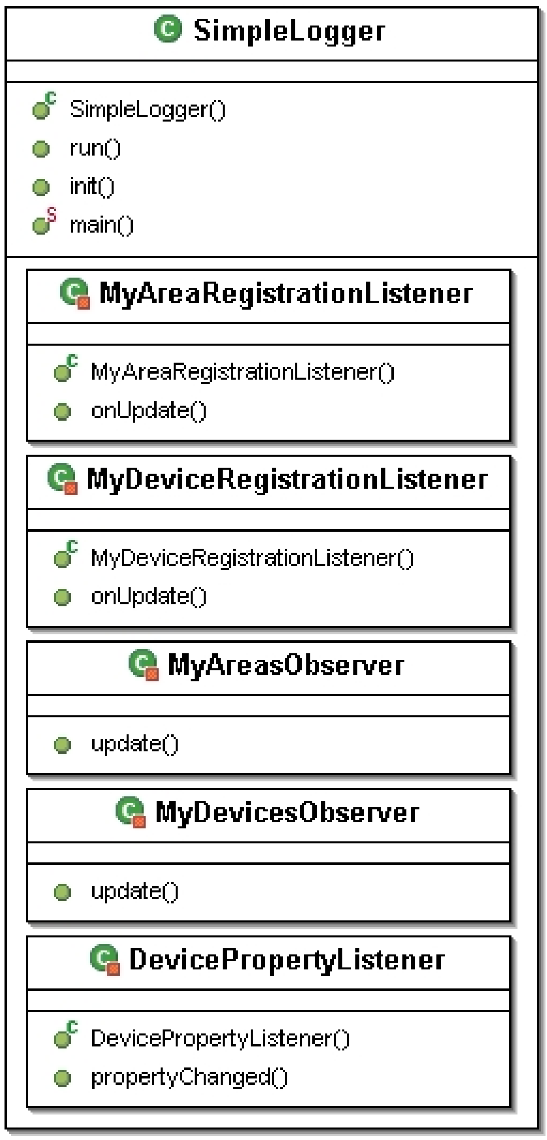


Figure 9.1: SimpleLogger UML

9.1.2 Code inspection

As every example the `ABISystem` needs to be initialized (See section: 8.1.2). Additionally though following initializations are necessary (Line 3 and 4). In order to be notified about any 'Adds' or 'Removes' the listeners need to be implemented and registered to the `ABISystem`.
When areas are changed i.e. when the location of a device has been altered by an other client application, we need to know about this incident. We achieve this to register an `Observer` for each area. In this example we map one observer to several areas of course (Line 8-11).
Further since we want to be responsive about any device alterations we need to implement and register an `Observer` as well (Line 13-20). What is the purpose of tracking devices when not being able to track changing device properties? Line 19 counteracts this issue by registering a `DevicePropertyListener` for each device currently accounted within the ABI System.

```
1 public void init() throws ConnectException, DeviceRegistrationException
2 {
3     system.addAreaRegistrationListener(new MyAreaRegistrationListener());
4     system.addDeviceRegistrationListener(new MyDeviceRegistrationListener());
```

```java

    Area[] areas = system.getAllAreas();

    for (int i = 0; i < areas.length; i++)
    {
        areas[i].addObserver(areasObserver);
    }

    BusMultiplexer multi = system.getMultiplexer();
    Device[] dev = multi.getAllDevices();

    for (int i = 0; i < dev.length; i++)
    {
        dev[i].addObserver(devicesObserver);
        new DevicePropertyListener(dev[i]);
    }
}
```

When implementing a `DeviceRegistrationListener` you need to implement the `onUpdate` method (Line 5).
Specifically speaking when for instance a new device has been added by an other client application, the `onUpdate` method will be called. In order to figure out which device has been added by the ABI System, you might need to re-obtain the list of all devices currently known by the system by simply invoking.

```java
registeredDevices = system.getMultiplexer().getAllDevices();
```

By keeping track of `oldRegisteredDevices` (Line 14) you can find out which of the devices have been either added or removed. Note when a device has been added you might need to consider of registering the provided `Observer` called `devicesObserver` and a new `DeviceRegistrationListener` since you might want to receive device changes (device and property changes) in the near future as well. This applies to a device that has been removed from the ABI System too. So accordingly delete the `Observer` of the old device since it does not exist anymore.

```java
1  private class MyDeviceRegistrationListener implements DeviceRegistrationListener
2  {
3      //...
4
5      public void onUpdate()
6      {
7          //...
8          // When a device has been added
9          registeredDevices[i].addObserver(devicesObserver);
10         new DevicePropertyListener(registeredDevices[i]);
11
12         //...
13         // When a device has been removed
14         oldRegisteredDevices[i].deleteObservers();
15     }
16 }
```

The detailed code for this example can also be found at: http://www.ini.unizh.ch/∼snufer.

9.1.3 Running the example

Before running this example, you also might want to observe the messages that are being sent between the participants. So you can do this as explained in the previous chapters (See section: 8.1.3).

Having once connected to the server you can start the application and you will be able to log any traffic that the logger has signed up for.

For the appliance of this application please note that this example is not intended for professional usage. It has been explicitly developed for this tutorial. However this application can be started in two ways. By providing a filename as a parameter, the application will log into the file instead of the regular standard output (See below).

```
public static void main(String[] args) throws IOException, ConnectException
{
    Writer out;
    ABISystem abiSystem;

    if (args.length > 0)
    {
        out = new FileWriter(new File(args[0]));
    }
    else
    {
        out = new OutputStreamWriter(System.out);
    }

    abiSystem = new ABISystem(SERVER_NAME, SERVER_PORT);
    SimpleLogger logger = new SimpleLogger(abiSystem, out);
    logger.start();
}
```

9.1.3.1 Compile and Startup

`cd` into the `src` directory of the SimpleLogger application.
An additional optional parameter can be provided that switches the output to a file instead of the console (As mentioned above).

- Windows
  -How to compile the application:

  ```
  javac -classpath ../../libs/RBC-API.jar;../../libs/RBCRemoteImpl.jar;. *.java
  ```

  -How to run the application.

  ```
  java -cp ../../libs/RBC-API.jar;../../libs/RBCRemoteImpl.jar;. SimpleLogger logfile.log
  ```

- Linux
  -How to compile the application:

  ```
  javac -classpath ../../libs/RBC-API.jar:../../libs/RBCRemoteImpl.jar:. *.java
  ```

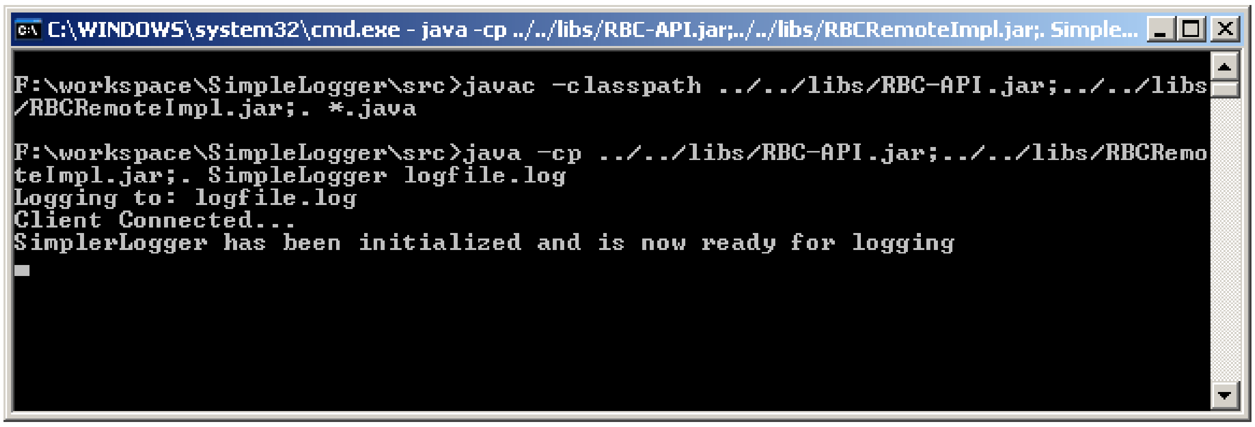
  -How to run the application:

  ```
  java -cp ../../libs/RBC-API.jar:../../libs/RBCRemoteImpl.jar:. SimpleLogger logfile.log
  ```

- Eclipse Platform
  Include the RBC-API.jar as explained in the beginning of the tutorial in section 6.2.

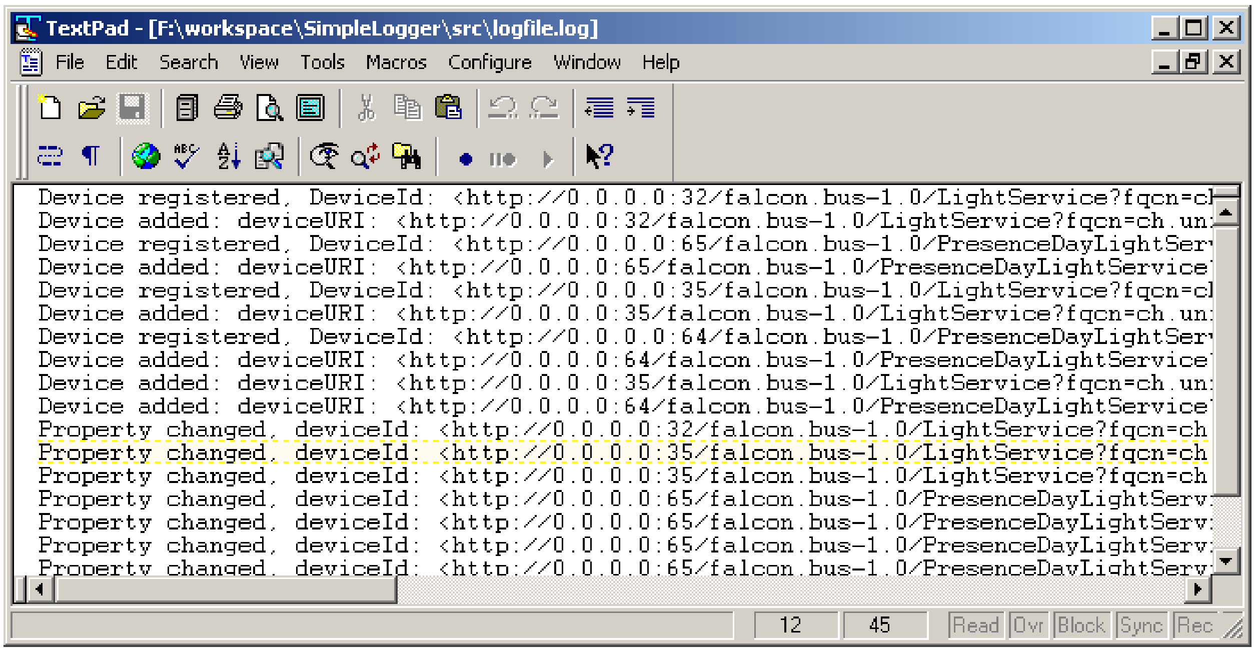
When the compilation and the startup were successful you should see following window (See figure: 9.2).

When having a glance into the logfile you will notice that gradually events are being logged (See figure: 9.3).


C:\WINDOWS\system32\cmd.exe - java -cp ../../libs/RBC-API.jar;../../libs/RBCRemoteImpl.jar;. Simple...

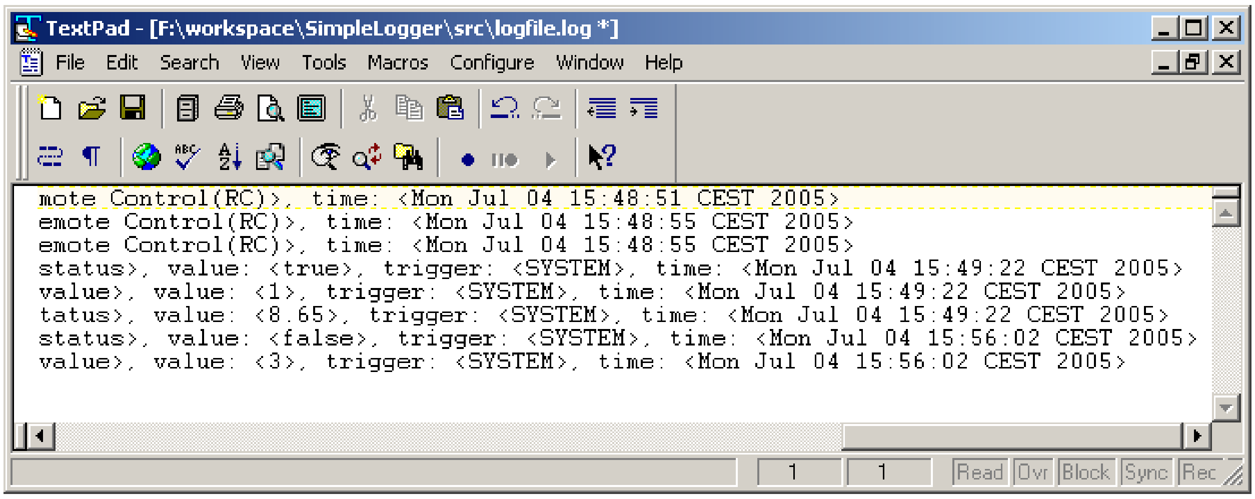
```
F:\workspace\SimpleLogger\src>javac -classpath ../../libs/RBC-API.jar;../../libs
/RBCRemoteImpl.jar;. *.java

F:\workspace\SimpleLogger\src>java -cp ../../libs/RBC-API.jar;../../libs/RBCRemo
teImpl.jar;. SimpleLogger logfile.log
Logging to: logfile.log
Client Connected...
SimplerLogger has been initialized and is now ready for logging
```

Figure 9.2: SimpleLogger Application


TextPad - [F:\workspace\SimpleLogger\src\logfile.log]
File Edit Search View Tools Macros Configure Window Help
12
45
Read Ovr Block Sync Rec

```
Device registered, DeviceId: <http://0.0.0.0:32/falcon.bus-1.0/LightService?fqcn=c
Device added: deviceURI: <http://0.0.0.0:32/falcon.bus-1.0/LightService?fqcn=ch.un
Device registered, DeviceId: <http://0.0.0.0:65/falcon.bus-1.0/PresenceDayLightSer
Device added: deviceURI: <http://0.0.0.0:65/falcon.bus-1.0/PresenceDayLightService
Device registered, DeviceId: <http://0.0.0.0:35/falcon.bus-1.0/LightService?fqcn=c
Device added: deviceURI: <http://0.0.0.0:35/falcon.bus-1.0/LightService?fqcn=ch.un
Device registered, DeviceId: <http://0.0.0.0:64/falcon.bus-1.0/PresenceDayLightSer
Device added: deviceURI: <http://0.0.0.0:64/falcon.bus-1.0/PresenceDayLightService
Device added: deviceURI: <http://0.0.0.0:35/falcon.bus-1.0/LightService?fqcn=ch.un
Device added: deviceURI: <http://0.0.0.0:64/falcon.bus-1.0/PresenceDayLightService
Property changed, deviceId: <http://0.0.0.0:32/falcon.bus-1.0/LightService?fqcn=ch
Property changed, deviceId: <http://0.0.0.0:35/falcon.bus-1.0/LightService?fqcn=ch
Property changed, deviceId: <http://0.0.0.0:35/falcon.bus-1.0/LightService?fqcn=ch
Property changed, deviceId: <http://0.0.0.0:65/falcon.bus-1.0/PresenceDayLightServ
Property changed, deviceId: <http://0.0.0.0:65/falcon.bus-1.0/PresenceDayLightServ
Property changed, deviceId: <http://0.0.0.0:65/falcon.bus-1.0/PresenceDayLightServ
Property changed, deviceId: <http://0.0.0.0:65/falcon.bus-1.0/PresenceDayLightServ
```

Figure 9.3: SimpleLogger Logfile


TextPad - [F:\workspace\SimpleLogger\src\logfile.log *]
File Edit Search View Tools Macros Configure Window Help
1
1
Read Ovr Block Sync Rec

```
mote Control(RC)>, time: <Mon Jul 04 15:48:51 CEST 2005>
emote Control(RC)>, time: <Mon Jul 04 15:48:55 CEST 2005>
emote Control(RC)>, time: <Mon Jul 04 15:48:55 CEST 2005>
status>, value: <true>, trigger: <SYSTEM>, time: <Mon Jul 04 15:49:22 CEST 2005>
value>, value: <1>, trigger: <SYSTEM>, time: <Mon Jul 04 15:49:22 CEST 2005>
tatus>, value: <8.65>, trigger: <SYSTEM>, time: <Mon Jul 04 15:49:22 CEST 2005>
status>, value: <false>, trigger: <SYSTEM>, time: <Mon Jul 04 15:56:02 CEST 2005>
value>, value: <3>, trigger: <SYSTEM>, time: <Mon Jul 04 15:56:02 CEST 2005>
```

Figure 9.4: SimpleLogger Logfile continued

Chapter 10

An AreaViewer Service

10.1 AreaViewer

The next example has rather been made for a demonstration purpose. Specifically speaking it has been developed to visualize the interaction between multiple tools such as the interaction between the ABI Admin ([NB05a]) and this one (AreaViewer).

10.1.1 Introduction

The AreaViewer is a graphical monitor of an area. It displays the states of all devices within this area and offers the opportunity to change the states of the effectors (See figure: 10.1). Predominantly this agent is used for observing areas. But as a further step it could be expanded to real control panel such as the ABI Admin ([NB05a]).

This tool arose of the necessity to have a possibility to visualize an area (i.e. a room) in an easy way. The AreaViewer provides a graphical interface that illustrates all devices currently added to a specific area. It is even capable to incorporate newly added devices, etc. in an ad-hoc fashion. The graphical user interface has been kept very simple, but it should give a general impression about the opportunities for any further implementations (i.e. The ABI Admin ([NB05a])).
In order to keep the AreaViewer as simple as possible, we only added the basic functions such as displaying the devices (i.e. Presence sensors, Lights and their effectors) and additionally to enable any device property alterations to be taken (i.e. switching on and off the lights), to it (Provided that each device can be altered in an either readable or writeable way ([NB05b])). An extra feature that has been implemented is an email notification message that can be sent to a destined person upon a device property change. For instance when the presence sensor suddenly becomes active an email can be sent to a given email address.

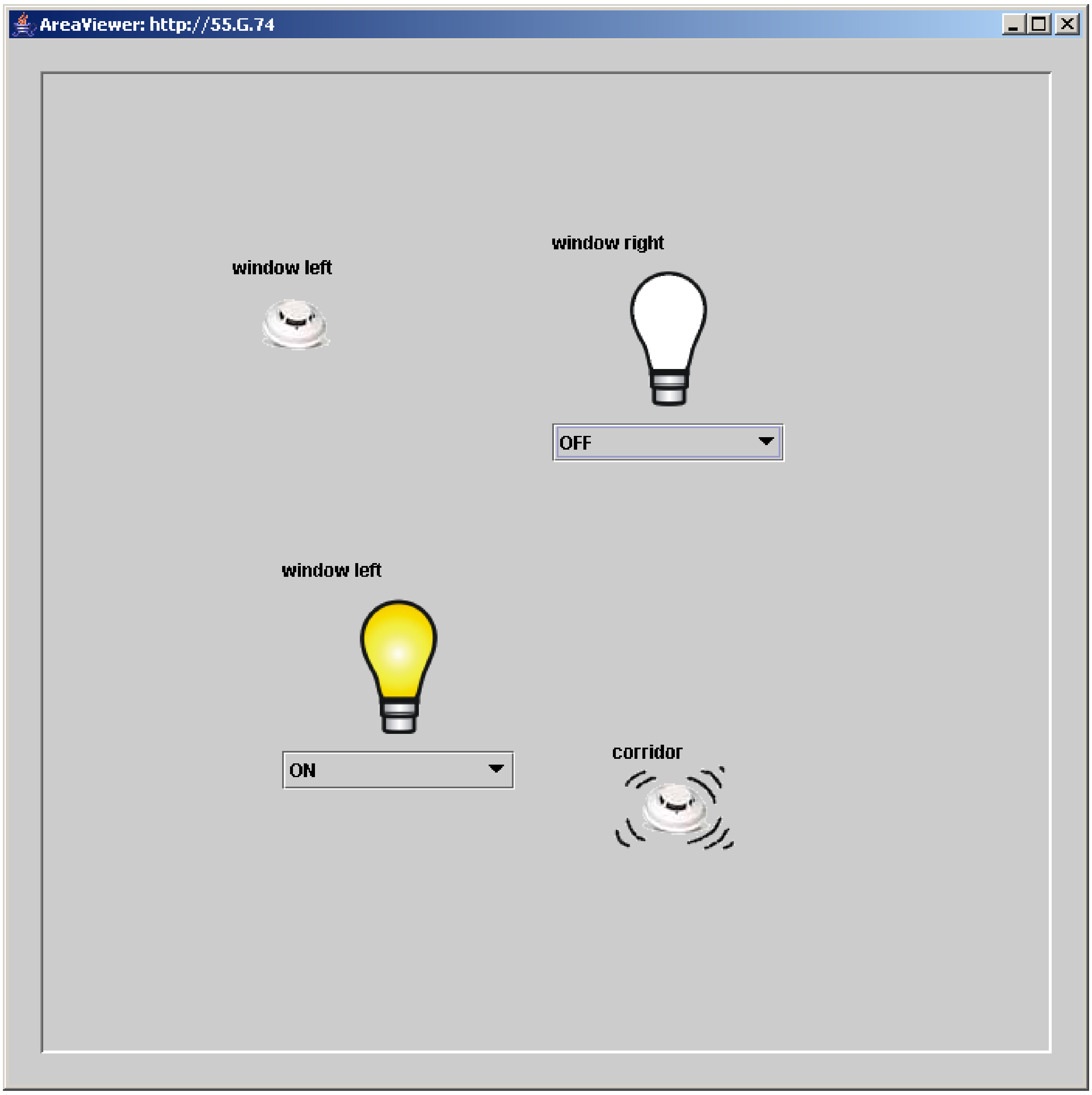


Figure 10.1: AreaViewer application

10.1.2 Code inspection

The central part of this application is the `AreaPanel class` (See figure: 10.2). The `AreaPanel class` displays all devices which are currenlty member of a specific area. Currently supported devices are presence sensors (`PresenceSensorPanel`) and lights (effectors) (`LightBulbPanel`).
Each devices type has been packed into a separate class and necessarily have been derived from a JPanel since we need a way draw the images such as a light bulb image on to such a panel (See below).

```
private class LightBulbPanel extends JPanel implements PropertyChangeListener
private class PresenceSensorPanel extends JPanel implements PropertyChangeListener
```

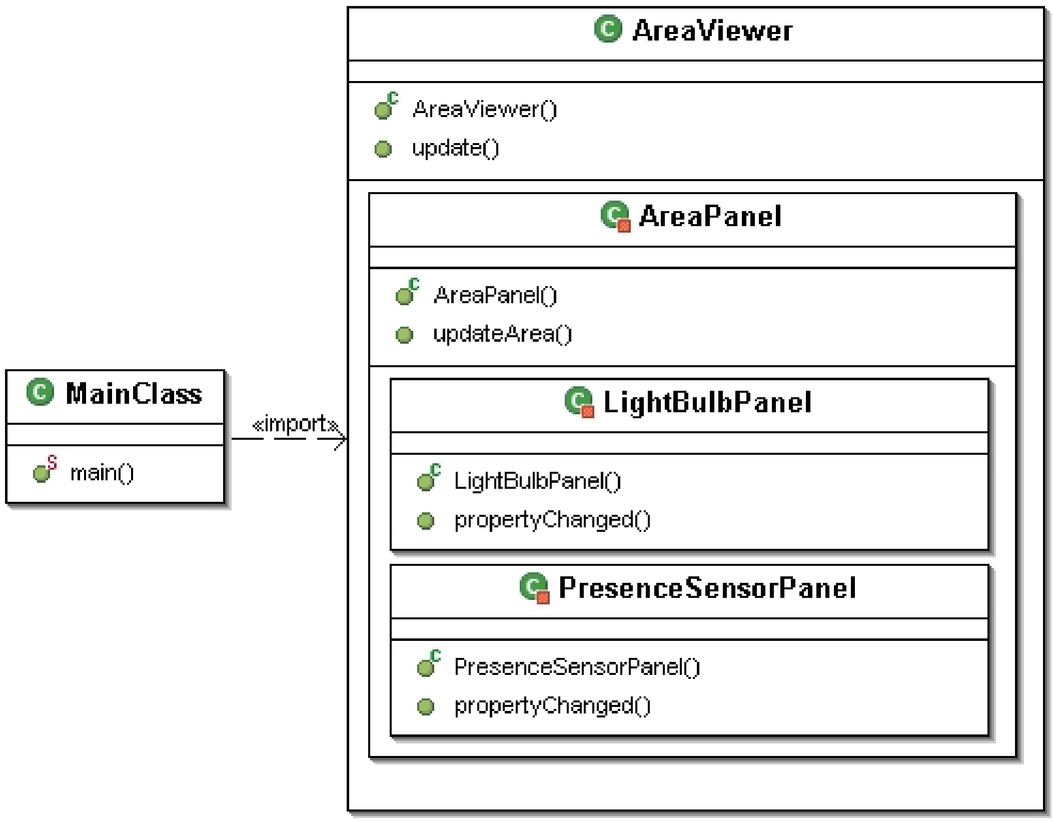


Figure 10.2: AreaViewer UML diagram

Because of the fact that both devices need to be notified upon device property changes, both need to implement the `PropertyChangeListener Interface`.

One other interesting part which might catch someones eye is when adding or removing devices to/from an area while observing the area with the AreaViewer.
With the help of the ABI Admin ([NB05a]) application, which is capable of administrating a whole bunch of things such as adding or removing devices to respectively from an area you shouldn't have any problems to test this case. As a result you will notice that the AreaViewer will adapt the new state and display the current area situation accordingly.

This is possible since the AreaViewer will be notified about any environmental changes of an area. Of course as we have seen in previous applications this is only possible when implementing an `Observer` (See below).

```java
public void update(Observable o, Object arg)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            if (areapanel != null)
            {
                 areapanel.updateArea();
                 areapanel.validate();
                 repaint();
            }
        }
    });
}
```

The method call `updateArea()` hereby updates the area in the sense that it needs to figure out if either new devices have added or existing devices have been removed by the ABI System (See below).
The methods `removeDeviceFromPanel()` and `addNewDeviceToPanel()` ultimately update the area panel accordingly.

```
public void updateArea()
{
    olddevices = devices;
    devices = thisarea.getAllDevices();

    // Remove old device from the panel
    for (int i = 0; i < olddevices.length; i++)
    {
        boolean toberemoved = true;
        // For all devices found in this area
        for (int j = 0; j < devices.length; j++)
        {
            if (olddevices[i].getDeviceURI().equals(
                    devices[j].getDeviceURI()))
            {
                toberemoved = false;
            }
        }
        if (toberemoved)
        {
            removeDeviceFromPanel(olddevices[i]);
        }
    }

    // Add new devices to the panel
    for (int i = 0; i < devices.length; i++)
    {
        boolean tobeadded = true;
        for (int j = 0; j < olddevices.length; j++)
        {
            if (olddevices[j].getDeviceURI().equals(
                    devices[i].getDeviceURI()))
            {
                tobeadded = false;
            }
        }
        if (tobeadded)
        {
            try
            {
                addNewDeviceToPanel(devices[i]);
            }
            catch (DeviceRegistrationException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

The `addNewDeviceToPanel()` (Line 41) method needs to create new devices and further creates an appropriate panel that illustrates one specific property. i.e. a *lightstatus* or a *presencestatus*. For each created property panel we must then locate a free space within the areapanel. This is accomplished with the `addOnFreeSpace()` method inside the `addNewDeviceToPanel()` method (Line 17).

```
1  private void addNewDeviceToPanel(Device device)
2          throws DeviceRegistrationException
3  {
4      // Read out all available properties
5      properties = device.getAllProperties();
6      for (int k = 0; k < properties.length; k++)
7      {
8          // Is a light
9          if (properties[k].getName().equals("lightstatus"))
10         {
11             Property lightstatusProperty = properties[k];
12
13             // Create LightBulb
14             LightBulbPanel lbp = new LightBulbPanel(
15                     lightstatusProperty, device);
16             devicePanels.put(device, lbp);
17             addOnFreeSpace(lbp);
18         }
19         //...
20     }
21 }
```

The detailed code for this example can also be found at: http://www.ini.unizh.ch/∼snufer.

10.1.3 Running the example

Before running this example, you also might want to observe the messages that are being sent between the participants. So you can do this as explained in the previous chapters (See section: 8.1.3).
Having once connected to the server you can start observing an area. Unfortunately you must configure this inside the code since we don't want to bother with cumbersome dialogs that would have been needed.

As stated in the introduction (See section: 10.1.1) this tool has been developed for a pure demonstration purpose only. So please note that this example is not intended for professional usage. It has been explicitly developed for this tutorial also.

10.1.3.1 Compile and Startup

`cd` into the `src` directory of the AreaViewer application.

- Windows
  -How to compile the application:

  ```
  javac -classpath ../../libs/RBC-API.jar;../../libs/RBCRemoteImpl.jar;. *.java
  ```

  -How to run the application.

  ```
  java -cp ../../libs/RBC-API.jar;../../libs/RBCRemoteImpl.jar;. AreaViewer
  ```

- Linux
  -How to compile the application:

  ```
  javac -classpath ../../libs/RBC-API.jar:../../libs/RBCRemoteImpl.jar:. *.java
  ```

  -How to run the application:

  ```
  java -cp ../../libs/RBC-API.jar:../../libs/RBCRemoteImpl.jar:. AreaViewer
  ```

- Eclipse Platform
  Include the RBC-API.jar as explained in the beginning of the tutorial on section 6.2.

When the compilation and the startup were successful you should see the allready known window (See figure: 10.1).

The tutorial is hereby seen as ended. For any other issues please consult the appropriate JavaDoc section (See chapter: 5.2) and for a further practical example you might also want to have a lock at the ABI Admin ([NB05a]) application.

Part V

Discussion and Future Work

Chapter 11

Discussion and Future Work

11.1 Overview

When basically detaching bundles into separate client applications we can observe following benefits and liabilities:

Centralised System Characteristics:

- One component with non-autonomous parts
- Component shared by users all the time
- All resources accessible
- Software runs in a single process
- Single Point of failure
- Single Point of control

Distributed System Characteristics:

- Multiple autonomous components
- Components are not shared by all users
- Resources may not be accessible
- Software runs in concurrent processes on different processors
- Multiple Points of control
- Multiple Points of failure

Certain common characteristics can be used to assess distributed systems:

- **Resource Sharing:** A Resource manager controls access, provides naming scheme and controls concurrency.
- **Openness:** Detailed interfaces of components need to be published.
- **Concurrency:** Components in distributed systems are executed in concurrent processes and the integrity of the system may be violated if concurrent updates are not coordinated.

- **Scalability:** Allow the system and applications to expand in scale without change to the system structure or the application algorithms.
- **Fault Tolerance:** Hardware, software and networks fail!
- **Transparency:** Distributed systems should be perceived by users and application programmers as a whole rather than as a collection of cooperating components. Enable local and remote information objects to be accessed using identical operations (*Access Transparency*).
  Enable information objects to be accessed without knowledge of their location (*Location Transparency*).
  Enable several processes to operate concurrently using shared information objects without interference between them (*Concurrency Transparency*).
  Enable multiple instances of information objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs (*Replication Transparency*).

We accomplished following stated common characteristics:

Characteristics	**Solution**
Resource Sharing	We definitely accomplished this point because we only provide one ABI System that actually acts as the resource manager.
Openness	The RBC Protocol and also the RBC API provide a stable standard that defines possible actions for distributed client applications.
Concurrency	Violation does not take place since we only provide one resource manager. The faster wins!
Scalability	As a matter of fact the scalability criteria is quite difficult to accomplish but since we going to have an option to transplant any distributed client application to a bundle as well (See section: 11.3) we can silently say yes. On the other hand though we must consider that the ABI System consums quite a chunk of CPU and memory as well. So the scalability factor might not one of the criteria that can be identified as accomplished. Still a distributed approach does still make the entire system more scalable as it would when only providing a single system component.
Fault Tolerance	Although fault tolerance is a criteria which can frankly speaking only accomplished by either providing recovery or redundancy, we achieve this criteria explained at section: 11.3.
Transparency	Transparency is definitely one of the criteria that we have been aspiring. We provide access, location, concurrency and in one way also replication transparency by applying the RBC API (See section: 11.2)

Table 11.1: Characteristics and their solutions

11.2 Discussion

When facing the fact that the ABI System has been fully implemented using bundles one might think that the adaptability and the extendability is 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 this wasn't true. 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 one can see the big mess we have inside. Especially with the usage of the Wireadmin which in turn is design technically a very good idea and concept but as a result the entire framework application gets that much of low coupled that barely anyone is capable of keeping track of the entire system anymore. The flip side to this coin is that bundles are praised to be developed independently and still require each other in an

invisible way.
Additionally image how such as system is being maintained by a semi-yearly shift of the developers...
Well we must confess that exaggeration was involved in the previous statements. But the previous system was that unstable that we practically needed to go through the entire code and partly even get rid of some bundles.
This was because the system hasn't been developed with the intention to be continued.
In short in the first part the entire ABI System has been refactored completely. As a second the RBC Protocol ([NB05c]) and the RBC API has been developed. With the applicability of the API we can reduce the system knowledge for other developers close to a minimum. Thereby we provide a common way of a programming API that most of the developers should be familiar with. Even though a OSGi Framework has been used one must know that 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.

11.3 Future Goal

This API strives for the goal of providing a third implementation of the RBC specification that is capable to support client applications to either be installed and launched as a separate bundle or as a distributed client application. Depending on the needs this might be a considerably huge advantage and might reflect the negative impacts such as overhead and uncertainty that a distributed system commonly brings along.

Nevertheless a distributed client applications still enjoys a huge advantage in all full strengths when considering non-functional requirements such as usability, testability and maintenance. A combination of the two can be particularly favorable in the development cycle. For instance to reduce the complexity and to advance the testability applications can be developed and tested completely independent. Without having to put up with the big OSGi Framework at all. Having once tested and completed the application, one can consider to switch the underlying reference implementation (See figure: 11.1) that installs and runs the application as a separate bundle inside the OSGi Framework.

However 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. For instance imagine an AI application that would suddenly stop controlling an area because of a network failure. Other disadvantages such as the network overhead have been identified above (See section: 11.1).

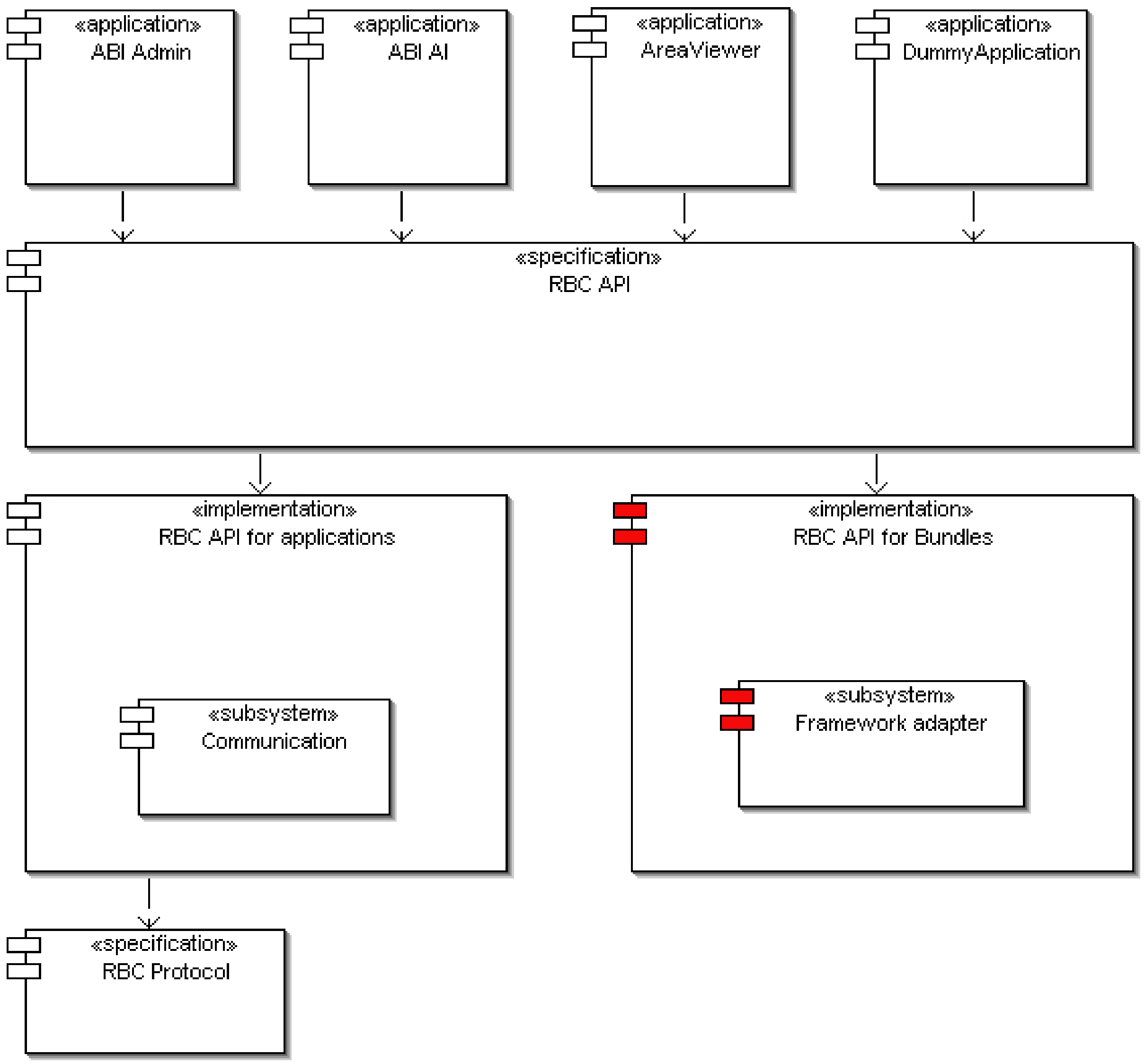


Figure 11.1: Future Architecture

Part VI

Glossary and Bibliography

Chapter 12

Glossary

Abbreviation	Explanation / Comment
ABI	Adaptive Building Intelligence
ABI System	The ABI System implemented using an OSGi Framework
Apache Axis	Apache Axis is an implementation of the SOAP
Bundles	Java JAR archive
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
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
RFC	Request for Comments
RMI	Remote Method Invocation (Java Technology)
RSI	Remote Service Invocation
Service	Interface exported by the service provider bundle
SOAP	Simple Object Access Protocol (W3C)
OSGi	Open Service Gateway initiative

Table 12.1: Glossary

Bibliography

[ECL] Eclipse. http://www.eclipse.org.

[ETH] Ethereal. a packet sniffer tool. http://www.ethereal.com/.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[JAVa] J2se. http://www.sun.com.

[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. Intelligent, learning system – a new abi system built on the open services gateway initiative. 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 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.

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

[PAC] Packet analyzer. a packet sniffer tool. http://www.networkchemistry.com/products/packetyzer.

Index