RBC Protocol specification

# Remote Building Control Protocol

**a protocol that is used to transfer building data**

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 part of the documentation is also part of the term project and has been used in the RBC Server [NB05a] and the RBC API[NB05b]. The protocol might be extended within the upcoming diploma thesis.

By all means this protocol is in its starting phase and is still being elaborated. When performing any alterations to the protocol would you please modify and add any changes accordingly? All the tools are provided with all the accompanying documentation.

Thanks. We hope that it helps. Good luck!

Stephan and Mathias.

## Abstract

The Remote Building Control Protocol (RBC) is an application layer protocol for distributed, collaborative, hypermedia information systems. The format of RBC describes a generic, stateless, protocol which can be used for many tasks beyond its use for building intelligence. When comparing HTTP with RBC one will notice that the header is built approximately the same way. In fact from the perspective of the HTTP ([HTT]) header, RBC has completely been resembled mostly following the guidelines of the HTTP header except that RBC still distinguishes between a body and a header part in the sense that its splits certain type of fields into a header and a body part. Therefore in contrast to other well-known protocols such as HTTP this protocol does not support the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. This protocol has been applied in games that simply exchange commands and necessary updates.

In this context the RBC Protocol defines additional parameters which need to be adhered by client and server API's. Particular features of such are: Non-blocking and blocking message calls. In the Context of the ABI System such messages are also know as Remote Service Invocation (RSI), since the ABI System itself is implemented using a OSGi Framework ([OSG],[KNO]) that is completely build up of services.

# 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 protocol to be used for exchanging information between the ABI System (RBC Server Bundle) and custom application programs that need guaranteed reliable transmission of data in a simple, ascii-based protocol. One major use of this protocol is to enable applications to retrieve changing device information and on the other hand commands which are executed in the RBC Server similar to remote procedure calls. Hereby it provides a standard that all ABI client applications need to adhere when communicating to the ABI System.
The reason why such a protocol is applicable in our context is although OSGi ([OSG]) has emerged into a very powerful and adaptive system framework, it also has its liabilities such as:

- High Complexity and ”hard-wired” dependencies
- Long period of vocational adjustment
- Difficult maintenance
- Complex testing and debugging methods

Those negative forces need to be circumvented in a way that it does not affect the hot-plugging capabilities. To counteract this issue we developed a distributed model 1.1 that makes use of this protocol that helps to separate server from any client concerns. The achievement is obvious. We want to provide the ability to develop any client system independently without having to put up with the OSGi Framework all the time. Specifically speaking we want to develop a system that can provide the capability to plug an AI-Client that takes control over the ABI System in a distributed fashion.

## 1.2 Document structure

This specification includes the following core section which describes the message format.

1. Introduction
2. Common RBC Message Packets
3. Control RBC Message Packets
4. RBC Bus Message Packets
5. RBC Device Message Packets

6. RBC Area Message Packets
7. RBC Update Message Packets
8. Appendix, Glossary and Bibliography

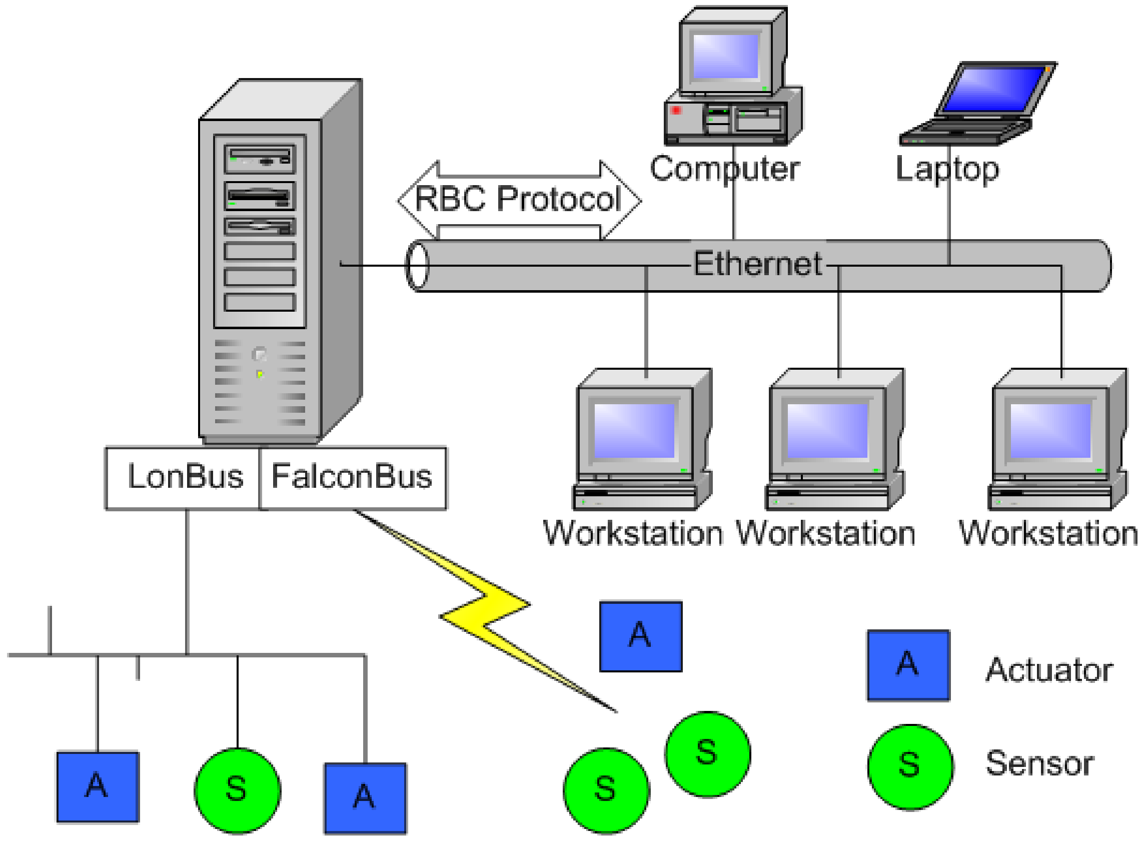


Figure 1.1: RBC Protocol

# Part II

# RBC Protocol Specification

# Chapter 2

# RBC Messages

## 2.1 General

For the remainder of this document, the requester is referred to as the client, while the target of the request is called the server.

RBC messages consist of requests from client to server and responses from server to client.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 ([RFCb]).

## 2.2 Overall Operation

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 Client-API and also the Server-API provides both blocking and non-blocking API calls. This can be easily done in an API level.
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 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.
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. Client gets this response in two ways. A good way, which has been applied in the reference implementation of the *RBCAPI* ([NB05b]) is some sort of callback. Specifically speaking the RBC is implemented using the dispatcher principle. Concretely this means that almost each message type that a client or the server is interested in has a corresponding message handler that is registered to a dispatcher. Hereby each message is processed by the competent handler upon dispatching.

The protocol used to send the RBC-message can be categorized mainly in to two types.

- Unidirectional: Client → Server OR Server → Client

- Bi-Directional: Client ↔ Server

## 2.3 Notational Conventions and Generic Grammar

Most of the mechanisms specified in this document are described in an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 ([RFCa]).

| Augmented BNF | Description |
|---|---|
| **name = definition** | The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names. |
| **"literal"** | Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive. |
| **rule1 \| rule2** | Elements separated by a bar ("\|") are alternatives, e.g., "yes \| no" will accept yes or no. |
| **(rule1 \| rule2)** | Elements separated by a bar ("\|") are alternatives, e.g., "yes \| no" will accept yes or no. |
| ***rule** | The character "*" preceding an element indicates repetition. The full form is "¡n¿*¡m¿element" indicating at least ¡n¿ and at most ¡m¿ occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two. |
| **[rule]** | Square brackets enclose optional elements |
| **implied *LWS** | The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token" below), since they would otherwise be interpreted as a single token. |

Table 2.1: Augmented BNF

## 2.4 Basic Rules

| Augmented BNF | Description |
|---|---|
| **OCTET:** | = ¡any 8-bit sequence of data¿ |
| **OCTETCR:** | = ¡any 8-bit sequence of data without CR and LF¿ |
| **CHAR:** | = ¡any US-ASCII character (octets 0 - 127)¿ |
| **CR:** | = ¡US-ASCII CR, carriage return (13)¿ |
| **LF:** | = ¡US-ASCII LF, linefeed (10)¿ |
| **SP:** | = ¡US-ASCII SP, space (32)¿ |
| **HT:** | = ¡US-ASCII HT, horizontal-tab (9)¿ |
| **<>:** | = ¡US-ASCII double-quote mark (34)¿ |

Table 2.2: Basic rules

RBC defines the sequence CR LF as the end-of-line marker for all protocol elements.

```
CRLF          = CR LF
```

Hexadecimal numeric characters are used in several protocol elements.

```
HEX            = "A" | "B" | "C" | "D" | "E" | "F"
               | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
```

Body field values may consist of words separated by LWS or special characters.

```
token          = 1*<any CHAR except CTL's or separators>
separators     = "(" | ")" | "<" | ">" | "@"
               | "," | ";" | ":" | "\" | <">
               | "/" | "[" | "]" | "?" | "="
               | "{" | "}" | SP | HT
```

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser.

```
TEXT           = <any OCTET and including LWS>
```

## 2.5 Message-Types

RBC messages generally consist of **requests** from client to server and **responses** from server to client. Unlike HTTP ([HTT]) this protocol allows a third one called **message**. Messages of type **message** can be sent from either the client or the server. The crucial difference between *message* to regular *request-response* messages is that it's implying a non-blocking one-way message.
All types of message consist of a message name and a message type, zero or more body fields, an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the body fields.
In the interest of robustness, servers SHOULD ignore any empty line(s) received where a message type or a message name is expected. In other words, if the server is reading the protocol stream at the beginning of a message and receives a CRLF first, it should ignore the CRLF.
The CRLF at the end of the last property indicate the end of the message and MUST be adhered.

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

Formally the message is defined as:

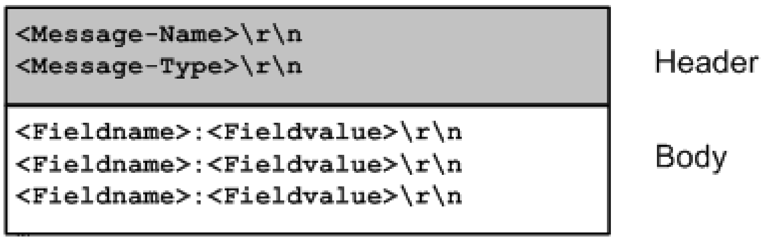
```
generic-message = messagename CRLF
                  messagetype CRLF
                  *(property CRLF)
                  CRLF
messsagename     = OCTETWCR
messagetype      = "Request" | "Response" | "Message"
```

### 2.5.1 Message Header

RBC header distinguishes from the HTTP header in the sense that it does not provide any version information. Basically the header consists of the following two fields which are followed by CRLF.

### 2.5.2 Message Body

RBC body fields, applies the same generic format as the header given by HTTP. Each header field consists of a name followed by a colon (":") and the field value. Field names are case-insensitive. The field value MAY be preceded by any amount of LWS, though a single SP is commonly not preferred.



```
<Message-Name>\r\n
<Message-Type>\r\n
<Fieldname>:<Fieldvalue>\r\n
<Fieldname>:<Fieldvalue>\r\n
<Fieldname>:<Fieldvalue>\r\n
```

Figure 2.1: The RBC Message Format

```
property       = field-name ":" field-value
field-name     = OCTETWCR
field-value    = OCTETWCR
```

## 2.6 RBC Message revisited

Having explained the basic protocol capabilities in a formal we can now define how such a packet (message) is structured in an informal way. We distinguish between three different type of messages: **Request, Response and Message**. The figure 2.1 illustrates the simply structured message format that need to be adhered by both parties (Client and Server). When performing blocking calls the clients creates a Request packet with an accompanying message-type(See 2.5). The Server will dispatch the message to the appropriate message handler that processes the message. When the server has completed all necessary tasks the server will send the appropriate Response message back to the client. When performing non-blocking calls the client simply structures a message of type **message**. Such kinds of messages are usually applied when sending commands which either do not provide any meaningful response message besides the common success response message. We can meet similar proceedings in the Apache Axis project ([Apa]). This also provides a blocking and a non-blocking API.

## 2.7 Defined messages

All fields which are currently specified by the protocol are now given regardless of their message type (request, response or message)

```
messagename   = reject
                | peer-id
                | disconnectclient
                | keepalive
                | showbuses
                | connectbus
                | disconnectbus
                | showalldevices
                | showdevices
                | regdevserv
                | deregdevserv
                | setproperties
                | getproperties
                | createarea
                | removearea
                | adddevtoarea
                | alterdevarealocation
                | remdevfromarea
                | showdevinarea
                | propertiesupdate
                | busupdate
                | deviceupdate
                | areaupdate
```

A detailed description of all valid field-values associated with one of those message names listed above will be given (See section: 2.8) or further by each specific message.

## 2.8 field-names and field-values

Certain field-names such as deviceurl, busid and areapid need a distinct format:

### 2.8.1 deviceurl

The deviceurl is used to identify a device. It is unique and is structured based on the "http" scheme. Unfortunately this schemes can currently not be documented since it has been adapted from the previous system. But it needs to be defined at a later point of time! Urgently! It's really is really messy!

## 2.9 Common RBC Message Packets

Common RBC Messages is a message packet category that defines special message packets which actually don't have anything to do with our context. However they provide necessary features which are quite valuable for a protocol to have when dealing with error afflicted messages and such.

### 2.9.1 reject message

Sometimes it is appropriate to reject messages which are not supported by the server and perhaps also by the client application. This protocol specification suggests sending reject messages to the corresponding station that initiated the request when the message format cannot be understood correctly.
Additional fields can provide information which indicates its cause. In the ABI System you observe reject messages when sending arbitrary (not supported) messages to the server station.

| | |
|---|---|
| **Message-Name:** | reject |
| **Scope:** | any |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | A reject message is usually thrown (sent) when the Message-Name (Handler or Command) is either not supported or not implemented and or registered. If the Client sends a reject message the server shouldl drop it since it does not reject, reject messages. Generally speaking though reject messages can be initiated by all participants but each of them don't necessarily need to be capable of processing the message. Therefore the scope has been stated as *any*. |
| **Body parameters:** | 1. request: Name of rejected message<br>2. reason: Reason of rejection<br>3. message: Additional field for any description |
| **Message example:** | see below |

```
reject
message
-----------------------------------
request:Some unknow message name
reason:Not supported message!
comment:-
```

### 2.9.2 return message

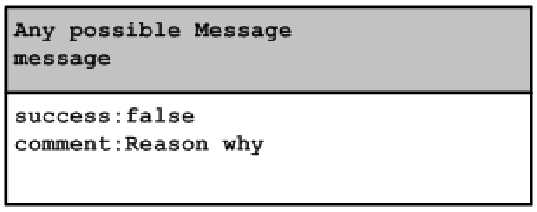
Messages are typically sent as request-response messages. Nevertheless it is still desirable to have the possibility to send the messages asynchronously. To support this feature all messages can be initiated and sent by a type called *message* which indicates a non-blocking message or call. In order to accomplish this feature, a non-blocking API needs to be implemented that fulfills this requirement. This applies for the client as well as the server API. Of course it's not meant to send all messages in the though of gaining time because the protocol does not provide any sequence numbers to match incoming and outgoing asynchronous calls again. Although it is possible to simply define some kind of handler that is called upon reception when using a non-blocking call.

The RBC API reference API implementation ([NB05b]) makes use of such a call when issuing an non-blocking call that retrieves all devices that can be found within the ABI System. This makes sense since it might take a while in order to retrieve all device information from the ABI System since each device can be hooked up to different buses which are possibly connected to different hardware and technologies and might cause different reaction time.
However this specification requires a non-blocking API that enables this kind of messaging.

| | |
|---|---|
| **Message-Name:** | return |
| **Scope:** | Server → Client |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | Each messages described below may provide a common return message... |
| **Body parameters:** | |

1. success: {true, false}
2. comment: Additional field for any description

**Message example:**



```
Any possible Message
message
success:false
comment:Reason why
```

## 2.10 RBC Control Message Packets

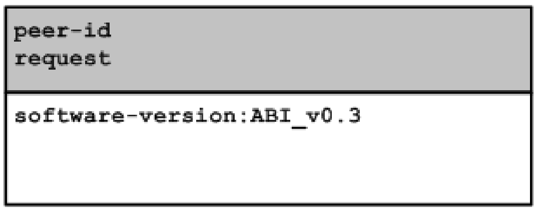
RBC Controlc Message Packets define all messages which have an associating context to control messages. Most application layer protocols support control packets that provide necessary information about clients or servers which i.e. provide remote peer information and such. So does this one. On one side ensure the compatibility and on the other to control the connection (establishment and termination). Other messages are to be included later such as the entire authentication procedure.

### 2.10.1 peer-id message

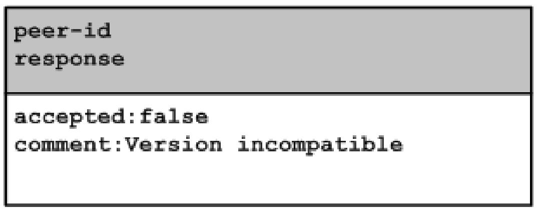
The peer-id message provides a simple check message between the participants. Although the protocol implementation MUST NOT include this feature since it would actually be the task of an entire authentication process mentioned above.

| | |
|---|---|
| **Message-Name:** | peer-id |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The peer-id is usually requested by the client and checked by the server for compatibility. |
| **Body parameters:** | 1. REQ, software-version: Client software version<br>2. RES, accepted:{true, false}<br>3. RES, comment:Additional field for any description |

**Request-Response example:**



```
peer-id
request

software-version:ABI_v0.3
```



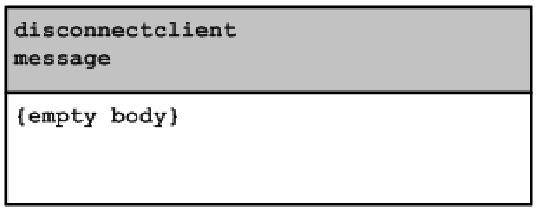
```
peer-id
response

accepted:false
comment:Version incompatible
```

### 2.10.2 disconnectclient message

Every client which holds a connection to the server has its own unique peer-id. Evidentially it is quite practical to provide a possibility to manually disconnect from the server. Technically it does do much but since most protocols provide this feature we added it as well.

| | |
|---|---|
| **Message-Name:** | disconnectclient |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The disconnectclient command is sent by the client upon termination. It has to be noted that this isn't a requirement. It's only a suggestion to be implemented as it helps the server to do cleanups by a predicting way. |
| **Body parameters:** | Empty body |

**Message example:**

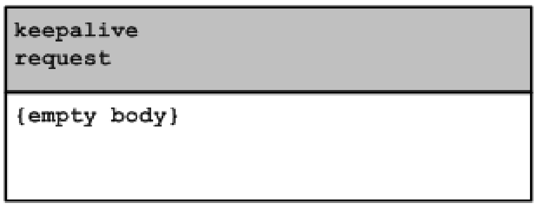


```
disconnectclient
message

{empty body}
```

### 2.10.3 keepalive message

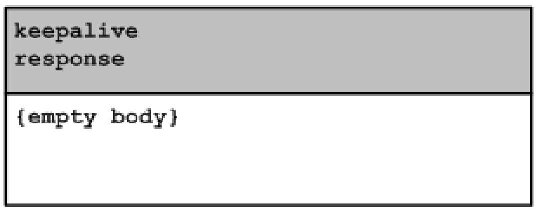
Every client needs to ascertain server availability. This is necessary since a client is always allowed to just listen for update messages (See section: 2.14) which would not be received upon a server or network problem. In order to counteract this issue a keep alive message is suggested.

| | |
|---|---|
| **Message-Name:** | keepalive |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The keepalive message command is sent by the client in order to determine server availablity. |
| **Body parameters:** | Empty body |



```
keepalive
request

{empty body}
```

**Request-Response example:**



```
keepalive
response

{empty body}
```

## 2.11 RBC Bus Message Packets (RSI)

RSI Messages (Remote Service Invocation Messages) are messages which are quite similar to Remote Procedure Calls (RPC)'s. In contrast to Remote procedure calls this protocol is capable of supporting a bidirectional communication channel that does not require any third party server installation (i.e. Apache Axis) or something like an rmiregistry as it is used in Java RMI. It is simple and well suited for the ABI System. Considering that all messages can be invoked synchronously (blocking) and asynchronously (non-blocking) as well.

### 2.11.1 showbuses message

The showbuses message is queried by the client with the intention of receiving all available buses currently know by the system. Hereby following status codes have been defined for this purpose:

| State | Description |
|---|---|
| **STATE_UNINITIALIZED = 0x00** | A bit value representing an uninitialized bus state, e.g. upon construction time. |
| **STATE_CONNECTED = 0x02** | A bit value representing a connected bus |
| **STATE_ATTACHED = 0x04** | A bit value representing a attached bus |

Table 2.3: Bus states

Whereas we can derive following combinations:

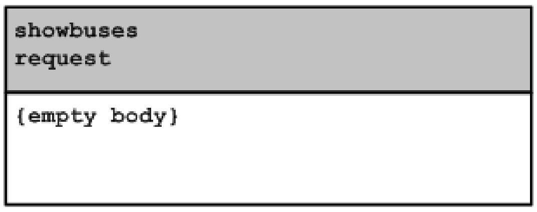
| State combination | Description |
|---|---|
| **STATE_CONNECTED & STATE_ATTACHED** | ⇒ Bus is available |
| **STATE_CONNECTED & ¬STATE_ATTACHED** | ⇒ Bus is unavailable |
| **¬STATE_CONNECTED & STATE_ATTACHED** | ⇒ Bus is attached but no connecting command was issued yet. |
| **¬STATE_CONNECTED & ¬STATE_ATTACHED** | ⇒ this bus was attached before, and now reconnect attempt will be taken when it becomes attached again. |

Table 2.4: Bus state combination

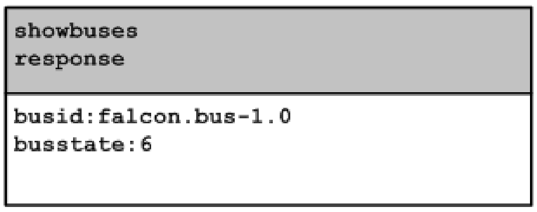
| | |
|---|---|
| **Message-Name:** | showbuses |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The showbuses command returns all available buses and their current state. Whether if they are connected or not. Client sends this message without any extra parameter(body). |
| **Body parameters:** | |

1. REQ, {empty body}
2. RES[1], busid: The unique bus identifier i.e. falcon.bus-1.0
3. RES[1], busstate: The state of the bus according to the previous description

**Request-Response example:**



```
showbuses
request

{empty body}
```



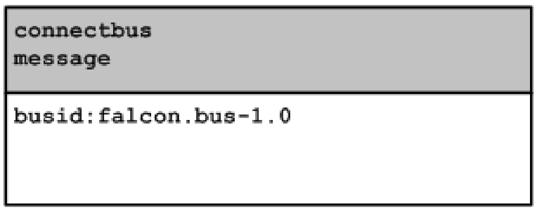
```
showbuses
response

busid:falcon.bus-1.0
busstate:6
```

### 2.11.2 connectbus message

| | |
|---|---|
| **Message-Name:** | connectbus |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The connectbus command connects a certain bus (busid) to the ABI System |
| **Body parameters:** | |

1. busid: The unique bus identifier i.e. falcon.bus-1.0

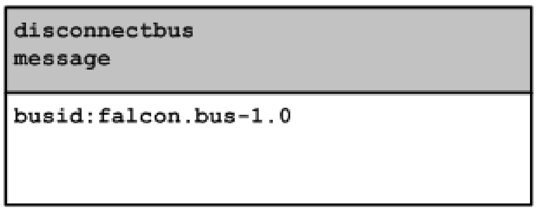
**Message example:**



```
connectbus
message

busid:falcon.bus-1.0
```

[1]For each bus that has been found a tuple of {busid,busstate} will be sent

### 2.11.3 disconnectbus message

| | |
|---|---|
| **Message-Name:** | disconnectbus |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The disconnectbus command disconnects a certain bus (busid) to the ABI System. All registered devices of that bus will hereby get unregistered. |
| **Body parameters:** | 1. busid: The unique bus identifier i.e. falcon.bus-1.0 |
| **Message example:** |  |

```
disconnectbus
message

busid:falcon.bus-1.0
```

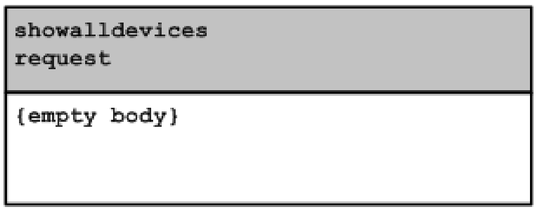
## 2.12 RBC Device Message Packets (RSI)

Device Messages Packets define all messages which have an associating context to devices. All messages defined here are messages which are initiated by the client as either of type request-response or message.

### 2.12.1 showalldevices message

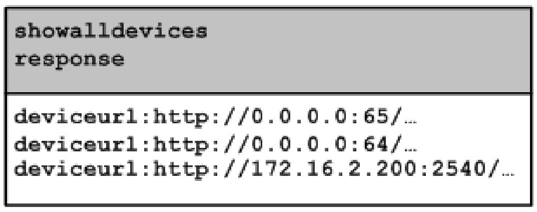
| | |
|---|---|
| **Message-Name:** | showalldevices |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The showalldevices command returns all registered devices(multiples)from the ABI System |
| **Body parameters:** | |

1. REQ, {empty body}
2. RES[1], deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...



```
showalldevices
request

{empty body}
```

**Request-Response example:**



```
showalldevices
response

deviceurl:http://0.0.0.0:65/…
deviceurl:http://0.0.0.0:64/…
deviceurl:http://172.16.2.200:2540/…
```

[1]Several devices maybe appended

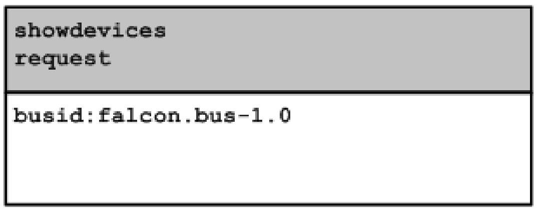
### 2.12.2 showdevices message

| | |
|---|---|
| **Message-Name:** | showdevices |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The showdevices command returns all registered devices (multiples) connected to a given bus from the ABI System. |

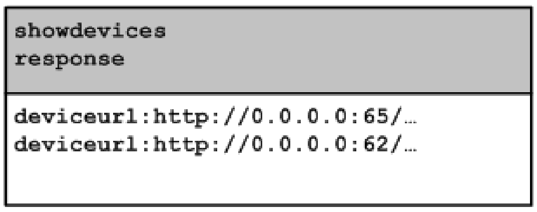
**Body parameters:**

1. REQ, busid: The unique bus identifier i.e. falcon.bus-1.0
2. RES[1], deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...

**Request-Response example:**



```
showdevices
request

busid:falcon.bus-1.0
```



```
showdevices
response

deviceurl:http://0.0.0.0:65/…
deviceurl:http://0.0.0.0:62/…
```

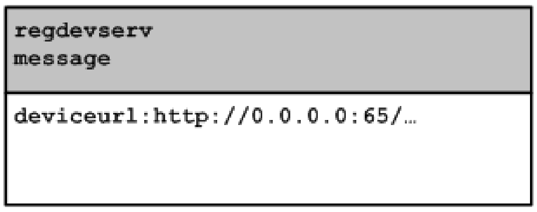
### 2.12.3 regdevserv message

| | |
|---|---|
| **Message-Name:** | regdevserv |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The regdevserv command registers a new device service. Commonly a devices service represents a physical device. Virtual devices are possible though. i.e. A software which is checking for presence. |

**Body parameters:**

1. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...

**Message example:**



```
regdevserv
message

deviceurl:http://0.0.0.0:65/…
```

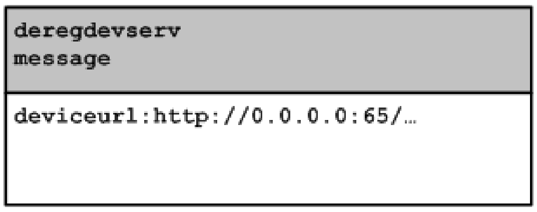
[1]Several devices maybe appended

### 2.12.4 deregdevserv message

| | |
|---|---|
| **Message-Name:** | deregdevserv |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The deregdevserv command deregisters an accounted device service. |
| **Body parameters:** | |

1. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...

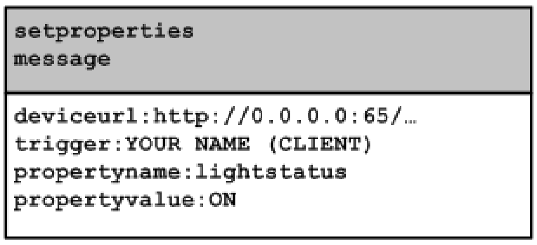
**Message example:**



deregdevserv message

deviceurl:http://0.0.0.0:65/…

### 2.12.5 setproperties message

| | |
|---|---|
| **Message-Name:** | setproperties |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The setproperties command can be initiated by three different type of persona. (triggers)<br>• Physical user i.e. Pressing light switch in a room<br>• AI (RBC Server)<br>• Client (Client ABI Software) authenticated by login<br>The command sets multiple device properties. i.e. A light can be either switched on or switched off; Blinds can be set into different positions. |
| **Body parameters:** | 1. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...<br>2. trigger: The originator of the message. (Name of the Client) i.e. The login name<br>3. propertyname: The custom property of a concrete device. i.e. A Falcon device<br>4. propertyvalue: The custom property value of a concrete device. i.e. A Falcon device |
| **Message example:** |  |

```
setproperties
message

deviceurl:http://0.0.0.0:65/…
trigger:YOUR NAME (CLIENT)
propertyname:lightstatus
propertyvalue:ON
```

According to the term-project documentation, every device does have its own set of well-defined properties. All writeable properties are authorized to be set by the client if desired. In order to submit changing properties such a message or command is necessary. The client-API uses this command when i.e. changing the lightstatus from ON to OFF or vice-versa.

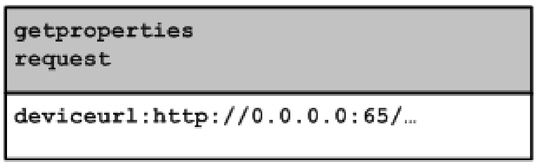
### 2.12.6 getproperties message

| | |
|---|---|
| **Message-Name:** | getproperties |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The getproperties command can only be initiated the client. The command requests all available properties of a device. i.e. A light can be either switched on or switched off or Blinds can be set into different positions. |

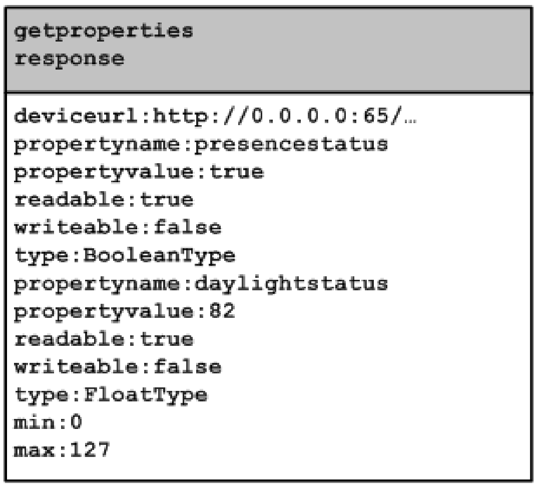
**Body parameters:**

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

**Request-Response example:**



```
getproperties
request

deviceurl:http://0.0.0.0:65/…
```



```
getproperties
response

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

[1]One single device may have several properties

According to the term-project documentation, every device does have its own set of well-defined properties.In order to query this information from these devices, this message or command is necessary. The client-API uses this command when obtaining detailed information about a device i.e. if the presence devices indicate presence or in order to retrieve the current measured daylight value.
Referring to the type constraint, this protocol defines 6 datatypes which can be seeing as a small IDL. Again, each type may have associating constraints which will be appended.
The value of the field type should match one of the datatypes listed in the first column of table 2.5.

| **Datatype** | **Constraint** |
|---|---|
| **EnumType** | list of enums: i.e.<br>enum:ON<br>enum:ON |
| **IntegerType** | {min, max} pair. i.e.<br>min:0<br>max:128 |
| **FloatType** | {min, max} pair. i.e.<br>min:0<br>max:128 |
| **DoubleType** | {min, max} pair. i.e.<br>min:0<br>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 2.5: Datatypes and their constraints

The protocol specification suggests to implement an API that provides an interface that defines all "custom" datatypes 2.5 according to the RBC specification. All concrete datatypes would hereby need to implement to this interface. We suggest naming the datatypes according to the names specified in 2.5. Further we suggest of providing a validation scheme in form of a method that checks corresponding values. Note that this is necessary since every datatype 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 propertytype with a small middleware customized for this kind of purpose. It defines a small set of an IDL in order that we know what kind of properties, each property provides. 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 ([NB05a]).

## 2.13 RBC Area Message Packets (RSI)

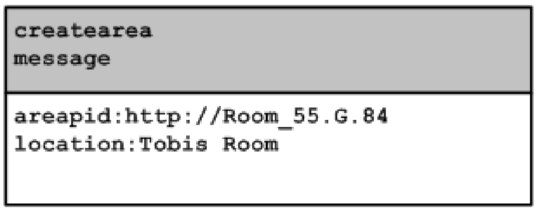
RBC Area Message Packets define all messages which have an associating context to areas. All messages defined here are messages which are initiated by the client as either of type request-response or message.

### 2.13.1 createarea message

| | |
|---|---|
| **Message-Name:** | createarea |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The createarea command creates a new area with a given location name. i.e. 55.G.74 |
| **Body parameters:** | |

1. areapid: An unique area identifier i.e. http://Room_55.G.74
2. location: The Name of the area i.e. 55.G.74

**Message example:**



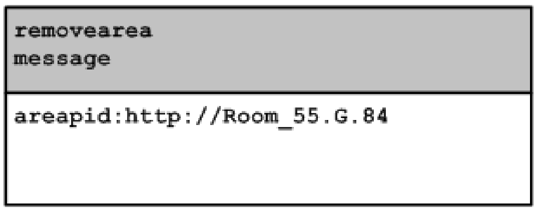
```
createarea
message
areapid:http://Room_55.G.84
location:Tobis Room
```

### 2.13.2 removearea message

| | |
|---|---|
| **Message-Name:** | removearea |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The removearea command removes an existing area from the ABI System |
| **Body parameters:** | |

1. areapid: An unique area identifier i.e. http://Room_55.G.74

**Message example:**



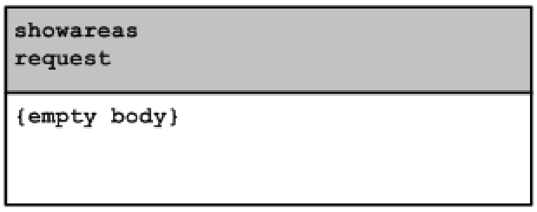
```
removearea
message
areapid:http://Room_55.G.84
```

### 2.13.3 showareas message

| | |
|---|---|
| **Message-Name:** | showareas |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The showareas command retrieves a list of all created command retrieves a list of all created (registered) areas from the the ABI System. |

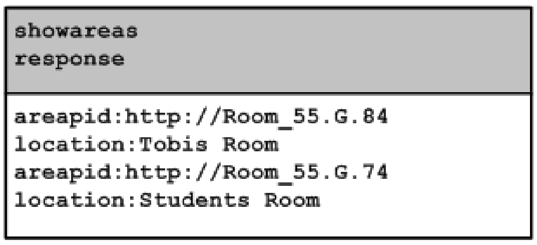
**Body parameters:**

1. REQ, {empty body}
2. RES[1], areapid: An unique area identifier.
   i.e. http://Room_55.G.74
3. RES[1], location: The name of the area
   i.e. 55.G.74



```
showareas
request

{empty body}
```

**Request-Response example:**



```
showareas
response

areapid:http://Room_55.G.84
location:Tobis Room
areapid:http://Room_55.G.74
location:Students Room
```

[1]The respond may contain multiple areas

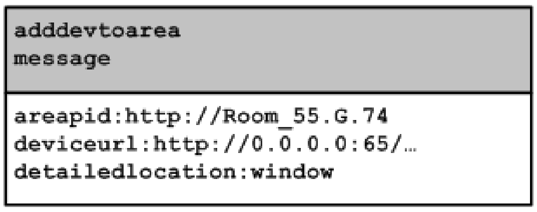
### 2.13.4 adddevtoarea message

| | |
|---|---|
| **Message-Name:** | adddevtoarea |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The adddevtoarea command adds a device with a given deviceurl to an existing area given by the areapid. |

**Body parameters:**

1. areapid: An unique area identifier i.e. http://Room_55.G.74
2. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...
3. detailedlocation: Provide a name where the device can be found or in which area the device is categorized. i.e. window

**Message example:**


```
adddevtoarea
message

areapid:http://Room_55.G.74
deviceurl:http://0.0.0.0:65/…
detailedlocation:window
```

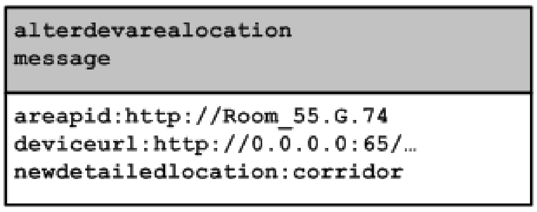
### 2.13.5 alterdevarealocation message

| | |
|---|---|
| **Message-Name:** | alterdevarealocation |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The alterdevarealocation command alters the detailed area location of an existing device's. |

**Body parameters:**

1. areapid: An unique area identifier i.e. http://Room_55.G.74
2. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...
3. newdetailedlocation: Provide a name where the device can be found or in which area the device is categorized. i.e. window

**Message example:**

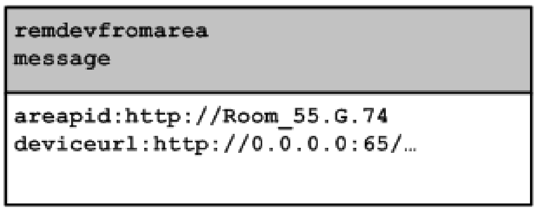

```
alterdevarealocation
message

areapid:http://Room_55.G.74
deviceurl:http://0.0.0.0:65/…
newdetailedlocation:corridor
```

### 2.13.6 remdevfromarea message

| | |
|---|---|
| **Message-Name:** | remdevfromarea |
| **Scope:** | Client → Server |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The remdevfromarea command removes a device with a given deviceurl from an existing area given by the areapid. |
| **Body parameters:** | |

1. areapid: An unique area identifier i.e. http://Room_55.G.74
2. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/... which area the device is categorized. i.e. window

**Message example:**



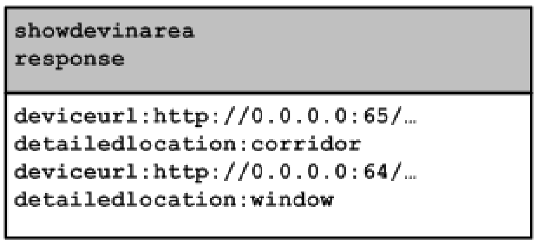
```
remdevfromarea
message

areapid:http://Room_55.G.74
deviceurl:http://0.0.0.0:65/…
```

### 2.13.7 showdevinarea message

| | |
|---|---|
| **Message-Name:** | showdevinarea |
| **Scope:** | Client → Server |
| **Common Message-Type:** | REQ → RES |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The showdevinarea command retrieves a list of devices currently accounted in an area given by the areapid an existing area given by the areapid. |
| **Body parameters:** | |

1. REQ, areapid: An unique area identifier i.e. http://Room_55.G.74
2. RES[1], deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...
3. RES[1], detailedlocation: The detailed location for this this i.e. window

**Request-Response example:**



```
showdevinarea
response

deviceurl:http://0.0.0.0:65/…
detailedlocation:corridor
deviceurl:http://0.0.0.0:64/…
detailedlocation:window
```

## 2.14 RBC Update Message Packets

RBC Update Message Packets define all messages which have an associating context to server updates. All messages defined here are messages which are initiated by the server and provided with the message type

[1]The respond may contain multiple devices

*message* only!

When having a glance back to figure 1.1 you will notice that with the event of time there will be quite some data sent to the server and to the clients. To clarify the issue a short example is given: After having successfully connected a bus (See section 2.11.2), you might want to register a couple of devices to the the ABI System (by issuing a regdevserv message defined in section 2.12.3). As soon as the registration has been completed, you would already receive device data upon any interactions with them. Either by manually pressing a light switch or when walking by a presence sensor that causes an event to be activated. It might be order of the day that a whole bunch of clients are interconnected with the RBC server simultaneously. Thus requiring to send updates of all kinds such as: Device registrations, deregistrations and other messages (See below) to all participants. Unfortunately they will notice any changes initiated by any device as well and theses are quite a few compared to the other messages mentioned before. Thus gradually causing many data packets sent by the server. Frankly speaking tough this is only a question of performance. -
Still, to counteract this issue a form of the publisher subscriber model has been defined. Every device which has been registered by any participating client needs a subscription. When clients want to be notified about any propertiesupdates 2.14.1 they need to provide a valid subscription to it before starting to receive any data of that device. Thus reducing the overhead generated by propertiesupdate messages
Note that other update messages don't need a subscription since busupdates 2.14.2, deviceupdates 2.14.3 and areaupdate 2.14.4 don't take place very often. Momentarily every client is authorized to perform any kind of subscription to any device available in the ABI System. To change this behavior additional messages need to be defined which actually fall into the RBC Control Message category.

### 2.14.1 propertiesupdate message

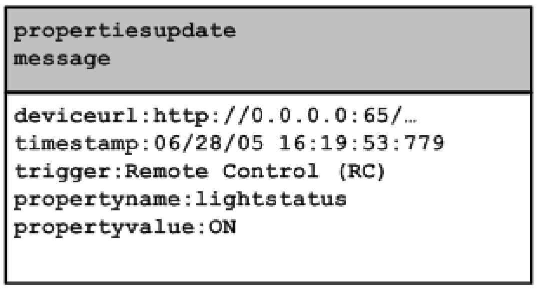
| | |
|---|---|
| **Message-Name:** | propertiesupdate |
| **Scope:** | Server → Client |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The propertiesupdate command can only be initiated by the server. The command broadcasts update messages which contain device properties of one device to all clients. The propertiesupdate command can be initiated by two different type of persons. (triggers) |

- Physical user i.e. Pressing light switch in a room
- AI (RBC Server)
- RBC Client (using setproperty message)

**Body parameters:**

1. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...
2. timestamp: The current time when the event has been triggered or activated [MM/DD/YY hh:mm:ss:S] (month/day/year hour:minute:second:millisecond)
3. trigger[1]:The originator of the message. (Name of the Client) i.e. login name
4. propertyname[1]: The custom property of a concrete device. i.e. A falcon device
5. propertyvalue[1]: The custom property value of a concrete device. i.e. A falcon device

**Message example:**



```
propertiesupdate
message

deviceurl:http://0.0.0.0:65/…
timestamp:06/28/05 16:19:53:779
trigger:Remote Control (RC)
propertyname:lightstatus
propertyvalue:ON
```

[1]The respond may contain multiple device properties

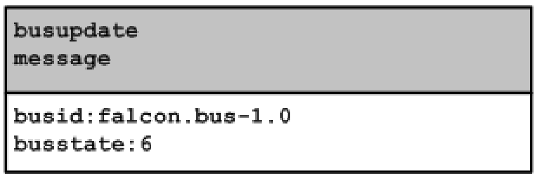
### 2.14.2 busupdate message

| | |
|---|---|
| **Message-Name:** | busupdate |
| **Scope:** | Server → Client |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The busupdate command can only be initiated by the server. The command broadcasts busupdate messages to clients. The contents is currently restricted to: busid and state.<br>The busupdate is typically sent by the server and initiated by other clients. |

**Body parameters:**

1. busid: The unique bus identifier i.e. falcon.bus-1.0
2. busstatus: The current busstate

**Message example:**



```
busupdate
message

busid:falcon.bus-1.0
busstate:6
```

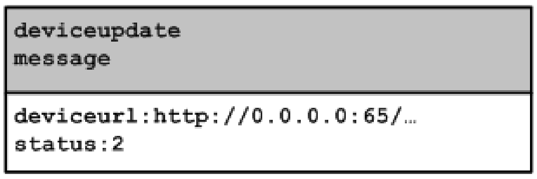
### 2.14.3 deviceupdate message

| | |
|---|---|
| **Message-Name:** | deviceupdate |
| **Scope:** | Server → Client |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The deviceupdate command can only be initiated by the server. The command broadcasts update messages to currently connected clients. It is important for a client to know when new devices are being plugged or removed from the ABI System. Either by another client or the system itself. That's why the protocol specification suggests to broadcast them along when being altered.<br>The busupdate is typically sent by the server and initiated by other clients. |

**Body parameters:**

1. deviceurl: The unique device identifier i.e. http://0.0.0.0:65/...
2. status: The current device status

**Message example:**



```
deviceupdate
message

deviceurl:http://0.0.0.0:65/…
status:2
```

| Device state | Description |
|---|---|
| **STATE_REGISTERED = 0x01** | A bit value representing that a device has been registered |
| **STATE_UNREGISTERED = 0x02** | A bit value representing that a device has been unregistered |

Table 2.6: Device states

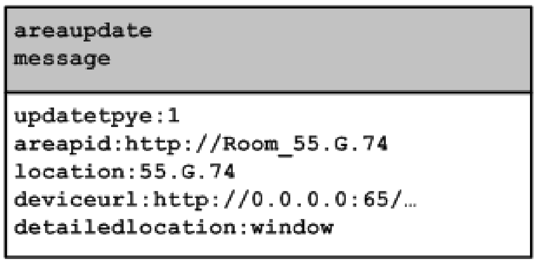
### 2.14.4 areaupdate message

| | |
|---|---|
| **Message-Name:** | areaupdate |
| **Scope:** | Server → Client |
| **Common Message-Type:** | Message |
| **Protocol Version:** | 1.0 |
| **Deprecated since:** | N/A |
| **Description:** | The areaupdate command can only be initiated by the server. The command broadcasts update messages to currently connected clients. It is important for a client to know when new areas are created, deleted from the ABI System. Either by another client or the system itself. An areaupdate includes all relevant data that must be know by all participants such as: All registered devices and their detailed location, the name of the area, etc. That's why the protocol specification suggests to broadcast them along. The areaupdate is typically sent by the server and initiated by other clients upon area creation,etc. |

**Body parameters:**

1. updatetype: States what has been altered, changed or deleted. updatetypes[1-5]
2. areapid: An unique area identifier i.e. http://Room_55.G.74
3. location: The name of the area
4. deviceurl[1]: The unique device identifier i.e. http://0.0.0.0:65/...
5. detailedlocation[1]: Provide a name where the device can be found or in which area the device is categorized. i.e. window

**Message example:**



```
areaupdate
message

updatetpye:1
areapid:http://Room_55.G.74
location:55.G.74
deviceurl:http://0.0.0.0:65/…
detailedlocation:window
```

| Update type | Description |
|---|---|
| **AREA_CREATED = 0x01** | A bit value representing that a new area has been created |
| **AREA_REMOVED = 0x02** | A bit value representing that a new area has been removed |
| **DEVICE_ADDED = 0x03** | A bit value representing that a new device has been added to the area |
| **DEVICE_REMOVED = 0x04** | A bit value representing that an existing device has been removed from the area |
| **DEVICE_LOCATION_CHANGED = 0x05** | A bit value representing that a the device location has been altered |

Table 2.7: Update-types

[1]Usually an area provides several devices which need to be retrieved

# Part III

# Glossary and Bibliography

# Chapter 3

# 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 |
| BNF | Backus-Naur Form |
| Bundles | Java JAR archive |
| HTTP | Hypertext Transfer Protocol |
| IDL | |
| 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 3.1: Glossary

# Bibliography

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

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

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

[NB05a] 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.

[NB05b] Stephan Kei Nufer and Mathias Buehlmann. Remote building control (rbc) api – an api that supports custom agent application to remotely control a building. Technical report, University of Applied Sciences Rapperswil, Switzerland and Institute of Neuroinformatics, Swiss Federal Institute of Technology, Zurich, Switzerland, 2005.

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

[RFCa] Rfc 822 (rfc822). http://www.faqs.org/rfcs/rfc822.html.

[RFCb] Rfc 2119 (rfc2119). http://www.faqs.org/rfcs/rfc2119.html.

# Index