Offene
Kommunikationssysteme
OKS

Fraunhofer Institute for Open Communication Systems

RESTAC Documentation

RESTAC Release: 1.1
Version of the Document: 20.12.07

Authors:

Murat Ates	<murat.ates@fokus.fraunhofer.de>
Philippe Bößling	<phib@cs.tu-berlin.de>
Stefan Föll	<foell@cs.tu-berlin.de>
Anna Kress	<anna.kress@fokus.fraunhofer.de>

Editors:

Ilja Radusch	<ilja.radusch@tu-berlin.de>
David Linner	<david.linner@tu-berlin.de>

Institute for Open Communication Systems
Technical University Berlin, Fraunhofer FOKUS

Abstract

RESTAC is a Java-based framework for REST-style interaction in distributed networks enabling the rapid implementation of peer-to-peer applications. It is based on a lightweight communication protocol built on HTTP, which makes it possible to easily address peers and access the functionality they provide through a web browser or any other HTTP client.

The basic layer of this implementation is a *communication layer based on HTTP*, which can be used for peer-to-peer communication, i.e. for peers to create requests to other peers, process such requests and finally reply to them. The layer provides TCP-based HTTP unicast messaging and UDP-based HTTP unicast and multicast messaging. On top of this, there is a *layer of resources*, which are bound to peers in tree-like structures. With the aid of a *wrapping model*, any Java object can be made available as a REST-style resource: the Java object and its methods are made available as remote services, with the state of the object being retrieved with GET or HEAD and modified with PUT or DELETE requests, and its methods executed with POST requests.

Abstract 2
1. Introduction 4
2. RESTAC Framework - Quick Start 4
3. Communication Layer 7
    3.1 Quick Start 7
    3.2 Technical Report 9
    3.3 Streams: Serialization and Parsing 12
4. Resource Model 14
    4.1 Common functionality of the resource model 14
    4.2 Server-sided functionality of the resource model 16
    4.3 Client-sided functionality of the resource model 18
5. Wrapping Model 19
    5.1 Server-sided functionality of the wrapping model 20
    5.2 Client-sided functionality of the wrapping model 22
References 24
Appendix A: Configuration 25
Acknowledgements 26

1. Introduction

REST is an architectural style for distributed, networked systems, based on the dissertation of Roy T. Fielding [1]. It is founded on the assumption that a network connects resources, whose status can be queried and modified. In a REST system the Hypertext Transfer Protocol (HTTP) can be used for communication, Uniform Resource Locators (URLs) for addressing and data formats like `text/plain`, `text/xml`, `image/gif` etc. for representation.

RESTAC is a Java based peer-to-peer framework implementing REST, which allows providing and making use of distributed services. Figure 1 shows an example of a small peer-to-peer network, as application in the domain of home automation, in which a user agent can query the state of a light, which can be controlled by a switch.

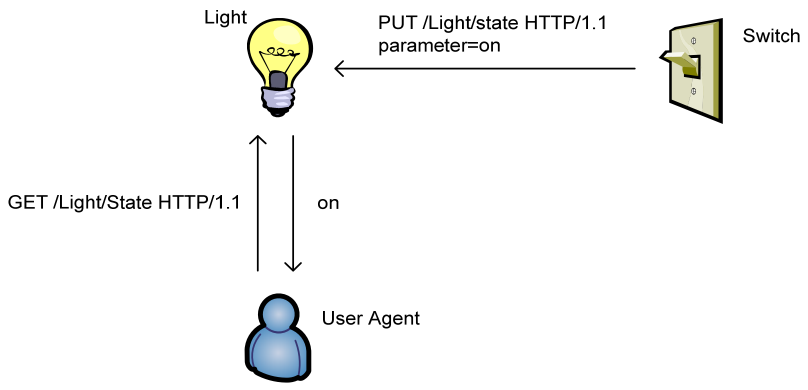


Fig.1 Example of a small peer-to-peer network

The basic layer of this implementation is a communication layer based on HTTP, which can be used for peer-to-peer communication, i.e. for peers to create requests to other peers, process such requests and finally reply to them. On top of this, there is a layer of resources, which are bound to peers in treelike structures. With the aid of a wrapping model, any Java object can be made available as resource. A basic Remote Procedure Call (RPC) like system allows the simple execution of distributed methods.

The rest of this document is organized as following: Chapter 2 gives an introduction into the basic usage of the framework, chapter 3 describes the package `de.fhg.fokus.restac.httpx` which builds the basis of the communication layer, chapter 4 describes the package `de.fhg.fokus.restac.resource.core` containing the functionality of the resource model, chapter 5 describes the package `de.fhg.fokus.restac.resource.wrapper` containing the functionality of the wrapping model.

2. RESTAC Framework - Quick Start

This chapter gives a short and compact introduction into the basic usage of the framework for the realization of peer-to-peer scenarios by describing the measures necessary for enabling a transparent communication between peers, notably a simple communication mechanism in the style of RPCs. Advantageously, the complexity of achieving this transparency for the realization of the scenarios shows to be rather low. A more detailed introduction into the whole functionality of the framework and explanation of the individual components can be found in the following chapters.

The effort necessary for the implementation of a peer-to-peer scenario is demonstrated in the following small exemplary application for controlling a switch. The state of the switch (`on={true,false}`) is supposed to be settable and gettable by all peers in the network. In a first step, a trivial implementation of a switch is given by the Java class `Switch`, which neglects (almost) all aspects of communication and can therefore be programmed as if for a local scenario. To control which methods are later exposed to other peers (and which not) and to be able to set some of the configuration parameters of the "Switch" resource Java annotations are used.

The sample code can be found in the package:

`de.fhg.fokus.restac.resource.wrapping.demo`

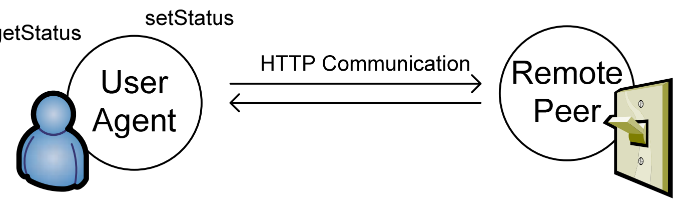


Fig. 3 Switch Remote Control

```
// Java class „Switch“ implements the functionality of the switch
@RestResource (path="Switch") // Here some config params can be set like
the path through which the resource will be accessed
public class Switch {

      private boolean on = false;

      @RestResourceMethod // this method will be exposed to other peers
      public boolean getStatus() {
                  return on;
      }

      @RestResourceMethod // this method will be exposed to other peers
      public void setStatus(boolean status) {
            on = status;
      }
}
```

After implementing the switch in Java, the subsequent steps show how to make it addressable by remote peers. In the following, these steps will be individually explained:

```
// The necessary classes from the framework must be imported

import de.fhg.fokus.restac.resource.wrapping.common.ObjectWrappingFactory;

public class SwitchWrappingTestClass {

      public static void main(String[] args) {

// In order to setup a new peer, an instance of ObjectWrappingFactory has
// to be created or to be available. During the instantiation of a factory,
// a HTTPX transceiver is automatically configured and started.

            ObjectWrappingFactory fac = new ObjectWrappingFactory();

// With the help of the registerObject() method of the ObjectWrapping-
// Factory the switch is registered under the URI “/Switch” (see Annotation
// of the Switch class), so that the provided services can be accessed by
// other peers using this URI.

            fac.registerObject(new Switch());
      }
}
```

Furthermore, the way a peer can access remote services is to be shown. To make it possible for the developer that local method calls are converted into requests on remote resources in the background, all operations are actually made on a proxy object locally representing the remote resource. In order to get such a proxy, a Java interface, which declares the remote methods, is needed. This approach promises more flexibility than the approach of other middlewares, which demand a static agreement about the common interface. Instead, components can be loosely coupled for a desired amount of methods. The individual steps for calling remote methods are examined in the following:

```
// Definition of the Java interface, which declares the desired methods for
// RPC communication
interface ISwitch {

      public boolean getStatus();
      public void setStatus(boolean status);

}
```

Now the peer can access the Switch resource:

```
public static void main(String[] args) {

// In order to setup a new peer, an instance of ObjectWrappingFactory has
// to be created or to be available.

      ObjectWrappingFactory fac = new ObjectWrappingFactory();

      try {


// With the help of the getObjectWrapper() method of the ObjectWrapper-
// Factory a proxy object for the resource which is accessible under the
// given URI is created, on which methods for remote
// communication can be called. Therefore, the returned proxy object has to
// be casted to the type which the given interface defines.

            URL urlForSwitch = new URL(HTTPXConstants.HTTP, "localhost",
2048, "Switch");

            ISwitch s = (ISwitch)fac.getObjectWrapper(urlForSwitch,
ISwitch.class);

// For the methods which are declared in the given interface, local method
// calls are executed remotely on the proxy. The necessary communication
// between local and remote peer is done transparently for the developer.
```

```
            s.getStatus(); // results in HTTP GET request

            s.setStatus(true); // results in HTTP PUT request

      } catch (MalformedURLException e) {
                  e.printStackTrace();
      }
}
```

A deeper introduction into the mechanisms of the Object Wrapping Model is given in chapter 6, which will also explain, how the method calls are mapped onto HTTP messages. At this place, only the most basic means necessary to make a Java object available as remote resource and to access the services it provides in an RPC style were to be portrayed.

3. Communication Layer

3.1 Quick Start

This Chapter gives a short and compact introduction into the basic usage of the communication layer of the RESTAC framework. Sample code can be found in **`package de.fhg.fokus.restac.httpx.demo.`**

```
// The necessary classes from the framework must be imported

import de.fhg.fokus.restac.httpx.core.dispatcher.HTTPXActionMessageDispatcher;
import de.fhg.fokus.restac.httpx.core.common.HTTPXActionMessageTransceiver;


// The starting point for a communication is the HTTPXActionMessageTransceiver. The
// config details of the transceiver are stored in the transceiver.properties file.
// A running (singleton) instance can be created/accessed by using
// getRunningInstance(). But it is also possible to start as many instances of the
// transceiver as necessary by using start().

HTTPXActionMessageTransceiver transceiver =
HTTPXActionMessageTransceiver.getRunningInstance();

// Depending on the direction of a communication, inbound for
// incoming and outbound for outgoing messages the appropriate dispatcher
// has to be used.

HTTPXActionMessageDispatcher inboundProxy = transceiver.getInboundDispatcher ();
HTTPXActionMessageDispatcher outboundProxy = transceiver.getOutboundDispatcher ();
```

Client side:

```
// To send out a HTTP request over the outbound proxy, an HTTPXActionMessage
// object has to be created first.
// All required attribute fields must be filled out.

Map<String, String> header = new HashMap<String, String>();
header.put("Host", "www.heise.de");
Path path = new Path("index.html");
```

```
ParameterList query = new ParameterList();
query.setParameter("KEY", "VALUE");

HTTPXActionMessage request = new HTTPXActionMessage("GET", "HTTP", "www.heise.de",
                                                    80, path, query, header, null);

// After creating the message, it is possible to send out and get
// back the response over the proxy.

HTTPXStatusMessage response = outboundProxy.deliverSynchronous (request);

// This example shows the asynchronous alternative.
// The class may implement the HTTPXStatusMessageHandler.

HTTPXStatusMessageHandle msgHandle = outboundProxy.deliverSynchronous (request);
msgHandle.addStatusMessageHandler(this);
```

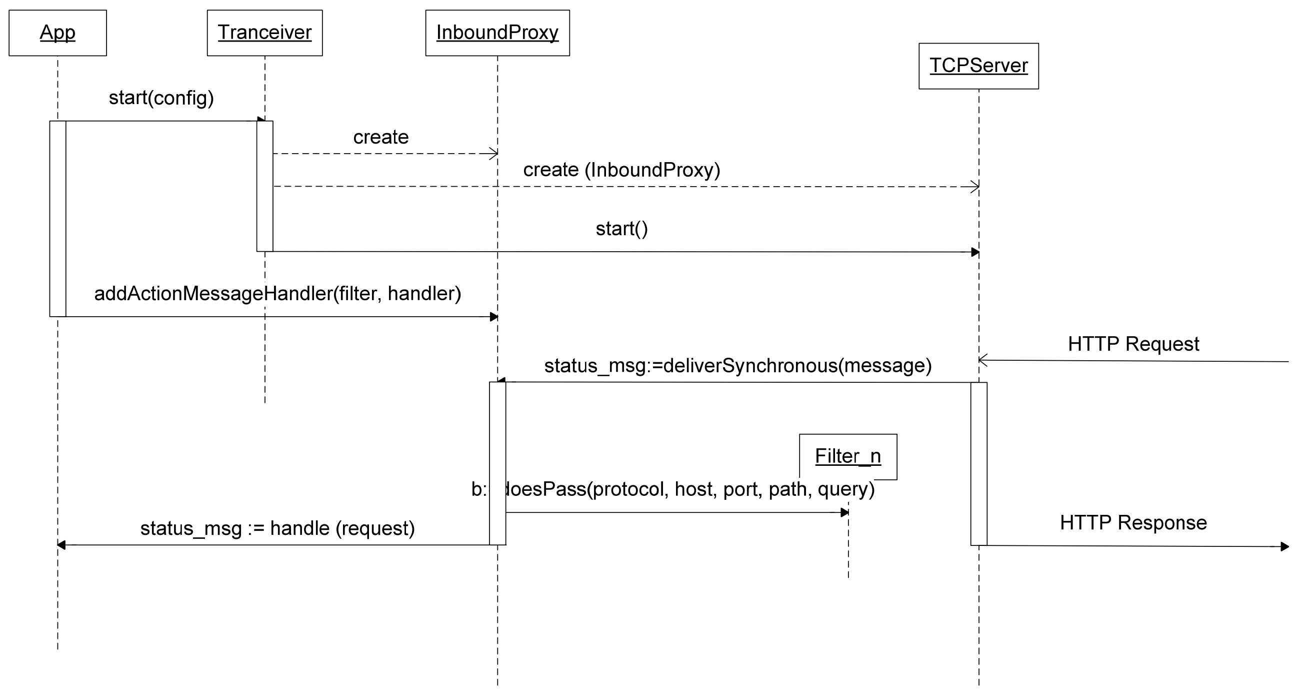
Server side:

```
// Class example implementing the synchronous server functionality.
// The class has to implement the interface HTTPXSynActionMessageHandler.

public class ServerApp implements HTTPXSynActionMessageHandler {

   public ServerApp(HTTPXActionMessageDispatcher inboundProxy){


// The ServerApp is registered as a handler for incoming messages with the inbound
// proxy. The handler will receive all messages with the parameters: protocol=HTTP,
// port:2221, path: myService/index.html.

      inboundProxy.addActionMessageHandler(new HTTPXActionMessageFilterImpl("HTTP",
                   null, 2221, new Path("myService/index.html"), null), this);
      }

      // Method for handling request messages and generating the response.
      public HTTPXStatusMessage handle(HTTPXActionMessage request){

             // process the request and generate the response message
             ...
             HTTPXStatusMessage response = new
                          HTTPXStatusMessage (200, "OK", "HTTP", header, content);

             return response;
      }
}
```

Finally, all registered `HTTPXManagedActionMessageDispatcher` can be shut down with the `shutdown`() method.

```
transceiver.shutdown();
```

This process is clarified in the figure.

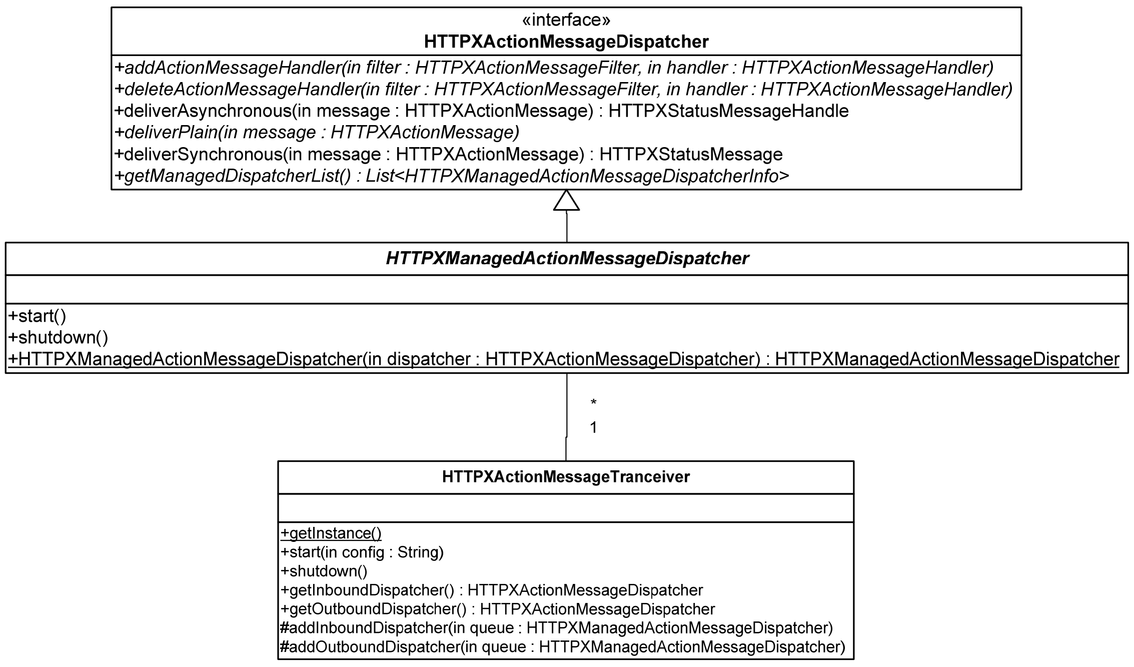

App
Tranceiver
InboundProxy
TCPServer
start(config)
create
create (InboundProxy)
start()
addActionMessageHandler(filter, handler)
HTTP Request
status_msg:=deliverSynchronous(message)
Filter_n
b: doesPass(protocol, host, port, path, query)
status_msg := handle (request)
HTTP Response

3.2 Technical Report

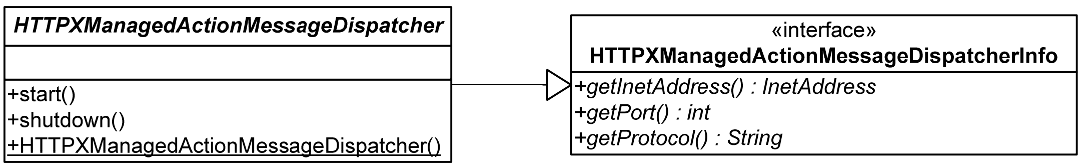
`package de.fhg.fokus.restac.httpx.core,`

`package de.fhg.fokus.restac.httpx.util`

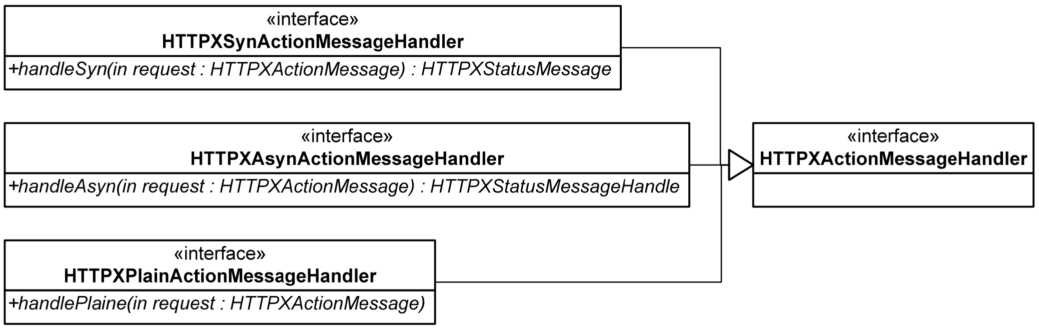
The starting point for a communication is the `HTTPXActionMessageTransceiver class`. Depending on the direction of a communication, it is possible to get an in- or an outbound proxy dispatcher as an instance of `HTTPXActionMessageDispatcher`. The `start(config)` method receives a String parameter with the name and the path of the property file in which every `HTTPXManagedActionMessageDispatcher` implementation is specified with its role `(e.g. Inbound[n]=...dispatcher.server.TCPServerManagedActionMessageDispatcher)`.

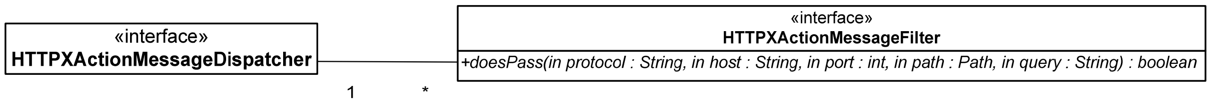

«interface»
HTTPXActionMessageDispatcher
+addActionMessageHandler(in filter : HTTPXActionMessageFilter, in handler : HTTPXActionMessageHandler)
+deleteActionMessageHandler(in filter : HTTPXActionMessageFilter, in handler : HTTPXActionMessageHandler)
+deliverAsynchronous(in message : HTTPXActionMessage) : HTTPXStatusMessageHandle
+deliverPlain(in message : HTTPXActionMessage)
+deliverSynchronous(in message : HTTPXActionMessage) : HTTPXStatusMessage
+getManagedDispatcherList() : List<HTTPXManagedActionMessageDispatcherInfo>
HTTPXManagedActionMessageDispatcher
+start()
+shutdown()
+HTTPXManagedActionMessageDispatcher(in dispatcher : HTTPXActionMessageDispatcher) : HTTPXManagedActionMessageDispatcher
*
1
HTTPXActionMessageTranceiver
+getInstance()
+start(in config : String)
+shutdown()
+getInboundDispatcher() : HTTPXActionMessageDispatcher
+getOutboundDispatcher() : HTTPXActionMessageDispatcher
#addInboundDispatcher(in queue : HTTPXManagedActionMessageDispatcher)
#addOutboundDispatcher(in queue : HTTPXManagedActionMessageDispatcher)

For each supported communication protocol an implementation should be available. Typical outbound dispatcher would be `TCPServerManagedActionMessageDispatcher` for TCP communication over server sockets, an `UDPUServerManagedActionMessageDispatcher` for UDP unicast over datagram sockets and a `UDPMUServerManagedActionMessageDispatcher` for UDP multicast over multicast sockets. These typical `HTTPXManagedActionMessageDispatcher` are bound by default to the ports `2048` (TCP), `1111` (UDP unicast), `2221` (UDP multicast). In case that these ports are already taken by other applications, different port numbers will be chosen by random. Every `HTTPXManagedActionMessageDispatcher` is automatically configured, started and shutdown over the `start()` and `shutdown()` method. Some information about running dispatcher is available through the `HTTPXManagedActionMessageDispatcherInfo` interface.

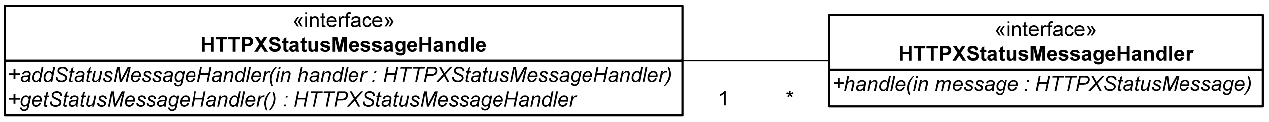

HTTPXManagedActionMessageDispatcher
+start()
+shutdown()
+HTTPXManagedActionMessageDispatcher()
«interface»
HTTPXManagedActionMessageDispatcherInfo
+getInetAddress() : InetAddress
+getPort() : int
+getProtocol() : String

The `HTTPXActionMessageDispatcher` proxy registers and manages `HTTPXActionMessageHandler` objects with a corresponding `HTTPXActionMessageFilter` to forward a request to the matching handler in order to process the request. `HTTPXActionMessageHandler` is a marker interface with no methods. One of the subclasses must be implemented. The `HTTPXActionMessageFilter` is also an interface with only one method (`doesPass(String protocol, String host, …)`). This method compares the registered filter with the incoming or outgoing request attribute fields, in case of accordance the registered handler will be called. Either an own filter implementation should be available or the default implementation (`HTTPXActionMessageFilterImpl`) of the framework is to use.

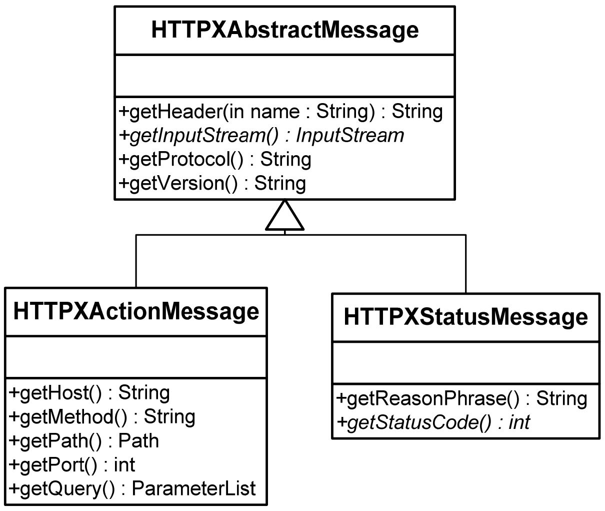

«interface»
HTTPXSynActionMessageHandler
+handleSyn(in request : HTTPXActionMessage) : HTTPXStatusMessage
«interface»
HTTPXAsynActionMessageHandler
+handleAsyn(in request : HTTPXActionMessage) : HTTPXStatusMessageHandle
«interface»
HTTPXPlainActionMessageHandler
+handlePlaine(in request : HTTPXActionMessage)
«interface»
HTTPXActionMessageHandler


«interface»
HTTPXActionMessageDispatcher
1
*
«interface»
HTTPXActionMessageFilter
+doesPass(in protocol : String, in host : String, in port : int, in path : Path, in query : String) : boolean

Now, it is possible to send out messages over the three methods `deliverSynchronous()`, `deliverAsynchronous()` and `deliverPlain()`. The `deliverPlain()` method has no return type, the usage makes only sense for UDP packages with no response message. The asynchronous handler returns an `HTTPXStatusMessageHandle` where the application adds an `HTTPXStatusMessage handler` in order to be informed if any response is available.


«interface»
HTTPXStatusMessageHandle
+addStatusMessageHandler(in handler : HTTPXStatusMessageHandler)
+getStatusMessageHandler() : HTTPXStatusMessageHandler
1
*
«interface»
HTTPXStatusMessageHandler
+handle(in message : HTTPXStatusMessage)

HTTP messages consist of a body, which contains the actual content of the message, and optionally several header fields with additional information about the message. The content is contained in an `InputStream`.



3.3 Streams: Serialization and Parsing

The packages `de.fhg.fokus.restac.httpx.util.streams` and `de.fhg.fokus.restac.httpx.util.serialization.*` provide comfortable tools for working with incoming and outgoing `InputStreams` and for serializing / parsing the content sent over the streams in HTTPX messages.

Since HTTP version 1.1 (compare RFC2616) the body of a HTTP message can be sent either “plain” or using chunked transfer coding. In the latter case the body is broken up into a number of chunks and each of the chunks is sent individually. Plain and chunked HTTP messages use different mechanisms to indicate the end of the message: Plain messages contain a Content-Length header where the length of the message is specified. Chunked messages do not have a Content-Length header, instead each of the chunks is prefixed with the according chunk length. Chunked transfer encoding is especially useful, if the length of the body is not known in advance because it is generated dynamically. If chunked transfer encoding is used, the Transfer-Encoding header of the message must be set to "chunked".

The package `de.fhg.fokus.restac.httpx.util.streams` provides two pairs of according classes for working with plain and chunked content for outgoing and incoming streams:

```
HTTPXPlainOutputStream / HTTPXChunkedOutputStream
HTTPXPlainInputStream / HTTPXChunkedInputStream
```

The stream classes assist the programmer in the setting of the correct (stream specific) headers and provide write and read methods for writing and reading raw byte arrays. The constructor of each output stream takes the content-type and the charset to be used for encoding as parameters. The constructor of each input stream takes the HTTPX status message as the only parameter and extract the necessary information for setting up the stream directly from the message.

Together with the package `de.fhg.fokus.restac.httpx.util.serialization.* the` contents of the streams can be read / written in a comfortable way. The package contains ContentWriter and ContentReader classes which provide write and read methods for content of different types. So far four types of content are supported: TextPlain, UrlEncoded, HtmlXml and TextXml. The write methods convert content into byte arrays which are then written into the streams (serialization). The read methods read byte arrays from the incoming stream and convert them into content (parsing). The programmer can extend the number of supported content types by implementing own ContentWriter and ContentReader classes. The constructors of the classes take the appropriate stream (either input or output stream) as their parameter.

After creating a stream the methods constructHTTPXActionMessage() or constructHTTPXStatusMessage() provided by the streams can be used which return an HTTPXActionMessage or an HTTPXStatusMessage respectively. The methods make sure that the appropriate headers for different types of streams (for example Content-Length in case of a plain stream) are set automatically. The headers parameter of the methods is a Map<String, String> which doesn´t have to be empty. Headers which were already put by the programmer into the Map and have nothing to do with the handling of streams are not overwritten, only the appropriate stream headers are “appended”, that means also put into the Map.

Simple example for writing content which is to be serialized using url-encoding:

```
HTTPXPlainOutputStream out = new
HTTPXPlainOutputStream(HTTPXConstants.TYPE_APP_URLENCODED,
HTTPXConstants.DEFAULT_CHARSET);

UrlEncodedContentWriter wr = new UrlEncodedContentWriter(out);

wr.write(content); // content is serialized and written into the stream
// construct request message with appropriate headers according to type of
stream
HTTPXActionMessage   request   =   out.constructHTTPXActionMessage("POST",
HTTPXConstants.HTTP, "localhost", 2048, new Path(path), query, null);
```

In case that the application doesn`t want to explicitly care about the type of the incoming stream (plain or chunked) a factory method can be used, which reads the headers of a message and returns the appropriate stream automatically. Example:

```
// get either plain or chunked input stream
HTTPXInputStream in = HTTPXStreamFactory.getHTTPXInputStream(message);

if(in.getContentType().equals(HTTPXConstants.TYPE_APP_URLENCODED)) {

// read and parse content of stream:
UrlEncodedContentReader rd = new UrlEncodedContentReader(in);

Map<String, String result = (Map<String, String>)rd.readBuffered();

}
```

Both, the streams and the writer / reader classes provide two types of read and write methods, “simple” (not buffered) and their buffered counterparts: read() and readBuffered() / write() and writeBuffered().

read() / write()
The difference between the “simple”, not buffered read() and write() methods for different types of streams (plain or chunked) is the following:
The write() method of a plain input stream simply writes the byte array into the stream “as it is”. The write() method of a chunked input stream takes the byte array, packages it into a chunk (adds chunk headers like chunk size) and writes it into the stream. **Attention:** care has to be taken if the programmer tries to write a chunk the size of which is greater than the provided maximum chunk size (the maximum chunk size can be set or a default value is used). In this case (due to an internal buffer) the method may block if the bytes are not immediately read out on the other side. To circumvent this problem the read() operation should be started in a thread if the programmer doesn’t (want to) make sure that the maximum provided chunk size is not exceeded. In case of plain input streams there is no such a problem, because the internal buffer is automatically set to the content-length of the message body and is therefore always big enough.
Equivalently, the read() method of a plain input stream simply reads and returns the byte array the length of which is indicated by the Content-Length header of the received message. The read() method of a chunked input stream reads the next chunk from the stream by inspecting the chunk headers which indicate its size, “unpackages” it and returns the content of the chunk.
The read() and write() methods of the writer / reader classes work in the same way on a higher conceptual level. They parse / serialize either the complete message body in case of a plain stream or chunks of data in case of a chunked stream.
readBuffered() / writeBuffered()

For plain streams the read() / write() methods and their buffered counterparts provide the same semantics (the buffered methods simply call the read() or write() method internally). Here the buffered methods were introduced only for the reason to keep the interface of both plain and chunked streams consistent.

For chunked streams the semantics are different yet:
The writeBuffered() method takes a byte array, splits it into chunks the maximum size of which can be set by the user (if it is not set a default value is used) and writes the series of chunks into the stream. The method can be useful if the user wishes to split a big array into chunks automatically which can be reassembled on the receiver side. The readBuffered() method of a chunked input stream respectively reads as long as there are chunks on the stream (the end is indicated by a zero size chunk), pastes them into one single byte array and returns the complete array.
The readBuffered() and writeBuffered() methods of the ContentWriter / ContentReader classes work in the same way on a higher conceptual level. They parse or serialize the complete content of the message by calling the readBuffered() / writeBuffered() methods of the streams.

4. Resource Model

`package de.fhg.fokus.restac.resource.core`

`package de.fhg.fokus.restac.resource.wrapping`

The resource model provides a set of basic abstractions for the handling of distributed resources. This way, the developer gets a simple, resource oriented view onto the underlying communication layer of the framework (see chapter 3). A resource in this context represents an entity, whose representation can be transported through the network and can be transformed.

As part of the resource model, the concept of an abstract tree is being introduced. With the help of a tree structure, a mapping of URLs onto the resources they address is created. Thus, the localization of resources can be logically separated from the access and the manipulation of their representations. The functionality of the resource model will be explained in the following.

4.1 Common functionality of the resource model

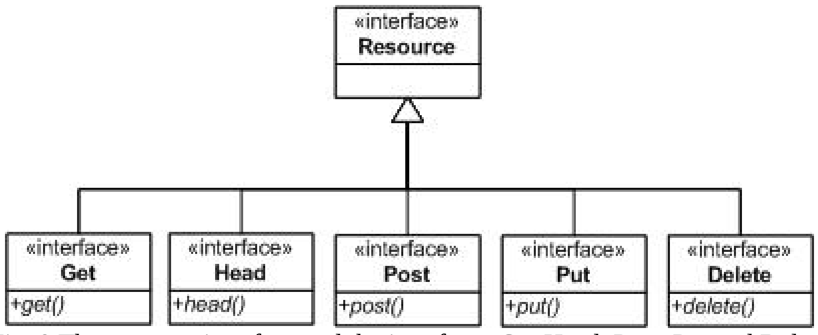


Fig. 9 The resource interface and the interfaces Get,Head, Post, Put and Delete connected to the respective HTTP methods

The model is designed so that every (distributed) resource is marked as such by implementing the interface `Resource`. Additionally, the interfaces `Get`, `Header`, `Post`, `Put` and `Delete`, which all extend `Resource`, can be used to define which HTTP methods the resource can process. By implementing these interfaces, the way manipulation shall be carried out can be defined for each resource. Examples would be the creation of new resources (though HTTP PUT/`Put`) or the retrieval of the representation of a resource (through HTTP GET/`Get`). However, the semantic of the respective HTTP method, as defined in RFC 2616 [4], applies for the usage of the interfaces.

The methods `post()` and `put()` (from the according interfaces) receive an object of type `HTTPXInputStream`. The `HTTPXInputStream` object represents the HTTP message body (compare RFC 2616). It is in the responsibility of the resources to parse the input stream. For parsing (and serialization) tools can be used from the package `de.fhg.fokus.restac.httpx.serialization`. These tools parse the input stream into a data structure specific to the content-type (e.g. `Document` for data type `text/xml`).

The methods `get()`, `head()` and `delete()` from the interfaces `Get`, `Head` and `Delete` do not receive any parameters. All the methods (including `post()` and `put()`) have the possibility though to access the parameters which are appended to the URL of the resource as query string in the received request. Especially getter-methods, on which resources implementing `Get` may map, may receive their parameters this way.

The methods `get()` and `post()` return an `HTTPXOutputStream` which is to be transmitted in the message body of the HTTP response. The HTTP response is constructed by the root manager of the resource tree which is presented below.

GET requests will receive a representation of the addressed resource as reply, while POST requests may receive some individual result. Requests using the other methods receive on successful execution a status message with an empty body, code 200 and text message “OK”.

On the server side, resources are organized and managed by a tree structure which is represented by the interface `Node`. Classes implementing this interface offer the functionality for traversing abstract trees with the aid of the methods `getChild()` and `getParent()`. To each node of the tree (except the tree root) a resource is connected.

This way, for each segment of the path of an incoming request an abstraction can be created in form of a node. Starting from the root, it is possible to navigate through the path over parent- and child nodes, on which in turn the request is mapped for processing. At each node the connected resource can be accessed with the `getResource()` method, on which the request can then be executed. The query string, which is to be appended to the URL of the resource, is passed on as a `ParameterList`.

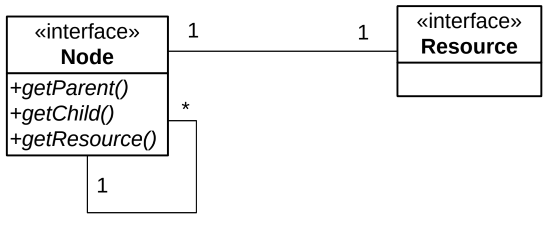


Fig. 10 1:1- Relation between node and resource

4.2 Server-sided functionality of the resource model

The class `ResourceMessageHandler` is an abstract class which implements the synchronous action message handler `HTTPXSynActionMessageHandler`. The class defines an abstract base class, which handles incoming requests and maps them on its managed resources according to the request path. In order to implement a resource tree a class has to extend the `ResourceMessageHandler` and implement the inherited abstract methods `getChild()` and `getResource()`. (A sample implementation is presented below.)

The class has to be registered with an inbound dispatcher by using a filter which especially will represent the path to the root of the tree. All requests starting with this path will be forwarded to the `ResourceMessageHandler` by the inbound dispatcher. By using `getChild()` for each segment of the path and subsequently `getResource()` the node and its connected resource which was addressed by a request can be determined by traversing segments of the path.

As described in the preceding chapter, the developer can provide application-specific code for the processing of the request by implementing the previously introduced interfaces (`Get`, `Head`, `Post`, `Put`, `Delete`). The actual call is handled by the code from the base class `ResourceMessageHandler`, though. For this, the path of the incoming request is being traversed, the request executed on the resource for the specified HTTP method, and if necessary, resource exceptions mapped on HTTP status codes. This includes the handling of special cases like when the processing of a request on the addressed path is not intended (`getChild()` returns `null`), or when the execution of the desired HTTP method is not implemented for this resource.

A simple example, in which a resource for the addition of numbers can be accessed at the path `/math/add` shall demonstrate an application of this abstraction. The example can be found in the package `de.fhg.fokus.restac.resource.demo`.

```
public class MyResourceMessageHandler extends
            ResourceMessageHandler {

      public MyResourceMessageHandler(HTTPXActionMessageFilter filter) {
            super(filter);
      }

      // Traversing departing from the root node to the node representing
      // <rootPath>/add
      public Node getChild(String name) {
            if (name.equals("add"))
                  return new MathNode();
            return null;
      }

      // There is supposed to be no resource connected to the path
      // <rootPath>
      public Resource getResource(ParameterList query) {
            return null;
      }

      // Node, on which the path <rootPath>/add is mapped
      public class MathNode implements Node {

            // Returns the parent node
            public Node getParent() {
                  return MyResourceMessageHandler.this;
```

```
		}

		// There shall be no nodes for further path segments
		public Node getChild(String name) {
			return null;
		}

		// Returns the resource connected to the path <rootPath>/add
		public Resource getResource(ParameterList query) {
			return new MathResource();
		}
	}

	public class MathResource implements Resource, Post {

		// Defines the URI of the resource to have the form
		// „<rootPath>/add“
		public String getUniformIdentifier() {
			return filter.getPath() + "/add";
		}

		// A request with HTTP POST shall result in an addition, all
		// other methods shall not be supported
		public InputStream post(InputStream in, int contentLength, 
String contentType) throws ResourceException, SerializationException, 
IOException {

			//get parameters from inputStream
			ContentContainer request = 
ContentContainerConverterTools.convertInstreamToContainer(in, 
contentLength, contentType);

			// do adding stuff for values contained in content
			Map<String, String> content = (Map<String, 
String>)request.getContent();

			Integer result = new Integer(0);

			for ( String number : content.values() )
			{
				result += new Integer(number);
			}

			// build response
			ContentContainer response = new 
TextPlainContent(result.toString());
			InputStream responseStream = 
ContentContainerConverterTools.containerToInstream(response);

			return responseStream;
		}
	}
```

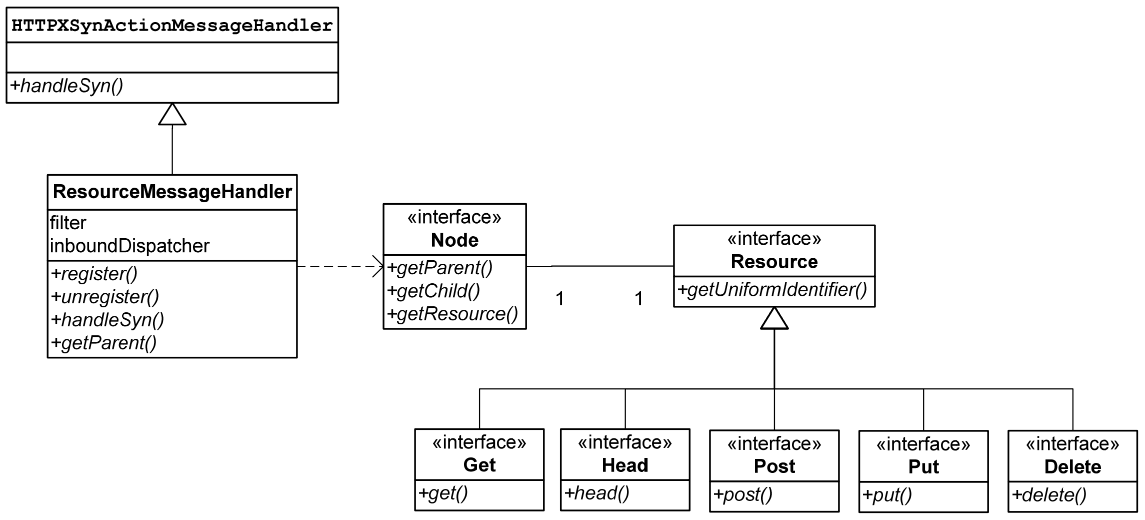


Fig. 11 Relevant classes and interfaces for the server-sided application of the resource model

4.3 Client-sided functionality of the resource model

With the help of the classes from the client package of the resource model it is possible to access remote resources and their functionality.

The class `NodeProxy` provides the basis for the application of the client-sided tree abstraction. For this, a node proxy is instantiated with a URL referencing a remote node and its connected resource. With the methods `getChild()` and `getParent()` the addressing can dynamically be changed relative to the base address, always returning a new node proxy. The initial path can be extended by a further token through the call of `getChild()`. The call to `getParent()` removes the last path segment. The node proxy instances on which the methods are called are not modified alongside, though.

For each node proxy the connected proxy for a resource (`ResourceProxy`) can be retrieved with the `getResource()` method. The remote resource is being represented locally by this instance. Calls of `get()`, `head()`, `post()`, `put()` and `delete()` on the local resource are sent as request to the actual remote resource making use of the appropriate HTTP method.

The static field `HTTPStatus.ALL_USED` contains all status codes for which a `ResourceException` is thrown after evaluation of the HTTP response. Method calls on a resource proxy must therefore be embedded in a `try-catch` block in order to handle eventually occuring resource exceptions.

A showcase call on a resource of another peer could look like the following:

```
HTTPXOutputStream out;
HTTPXInputStream out;

URL url=null;
try {
      // Defining the URL locating the remote resource
      url =new URL ("http://peer:port/math");
}
catch (MalformedURLException e) {}

// Creating a local node proxy
```

```
NodeProxy node = new NodeProxy(url);

// Navigation to the path representing the remote resource
node=node.getChild("add");

// Retrieving a proxy for the remote resource
ResourceProxy resource = (ResourceProxy)node.getResource(null);

// Creating the content which is to be contained as body in the
// request
out = … ;

try {
      // Call of post() to trigger an HTTP POST request to the
      // resource
       in = resource.post(out);
}
catch (ResourceException e) {
      // Handling of a response that did not return HTTP 200 OK, but
      // threw a resource exception, because of a status code from
      // Status.ALL_USED
}
```

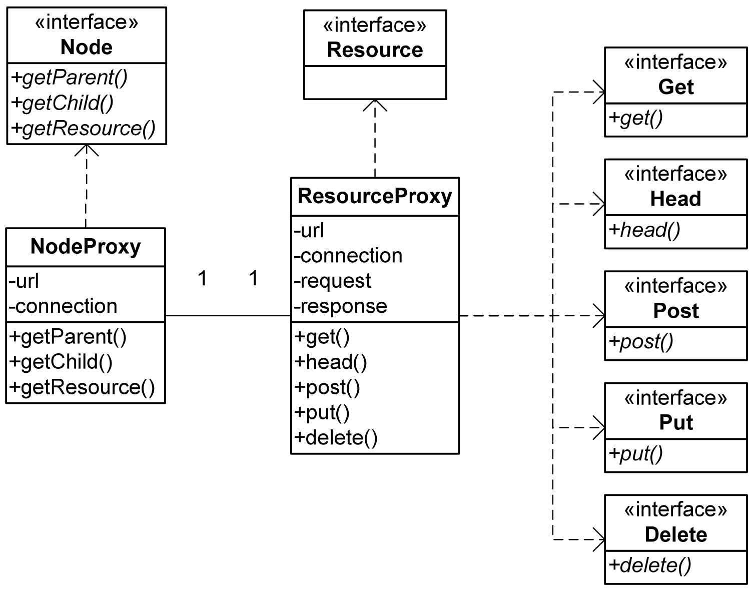


Fig. 12 Relevant classes and interfaces for the client-sided application of the resource model

5. Wrapping Model

`package de.fhg.fokus.restac.resource.wrapping`

Sample code is located in **`de.fhg.fokus.restac.resource.wrapping.demo`**

The wrapping model offers the possibility to make Java objects available as distributed resources, which can be manipulated through the methods they provide. For this, a mapping of the resource-based approach on top of an HTTP communication layer onto the object-oriented approach of Java with the help of Java Reflection [5] is introduced. In this context, resources are Java objects which can be addressed by HTTP requests. Depending on the HTTP method that is used and on the request path, one of the methods of the Java object's class is selected for execution. The parameters for the method call are thereby serialized and received url-encoded. To extend the wrapping model for previously unconsidered Java types, the classes ParserFunction and SerializerFunction have to be adapted, respectively (see Javadoc for already implemented types).

5.1 Server-sided functionality of the wrapping model

Java objects are made available as distributed resources with the help of a `RootWrapper`. The root wrapper itself extends the `ResourceMessageHandler` and realizes the mapping of Java method calls, together with the `InvocationWrapper`. In order to make a Java object available, it is passed to the constructor of the root wrapper, together with a specified filter on which it shall be registered:

```
// RootWrapper constructor:
RootWrapper(HTTPXActionMessageFilterResourceImpl filter, Object object)
```

For requests addressed to the root path of an object, e.g. `“/math”`, only the HTTP methods GET and HEAD are allowed. For any of the other methods a `“405 Method Not Allowed”` response is returned. For requests addressing resources relatively to the path on which the resource was registered, e.g. `“/math/add”`, a mapping on Java methods is made depending on the HTTP method that was used. This functionality is implemented in the class `InvocationWrapper`.

On HTTP GET and HEAD requests, a Java getter-method for the respective path token is searched for in the class of the wrapped object (e.g. on a request on `“/math/add”` a method `getAdd()` is searched for in the class of the bound object). GET requests return the result of the method call url-encoded.

On PUT and POST requests, on the other hand, a Java setter method for the respective path token is searched for in the class of the wrapped object (e.g. `setAdd(…)` for a request on `“/math/add”`). In the process, the parameters for the method call are parsed from the body of the HTTP request. In contrast to PUT requests, POST requests are handled in a way that at first Java methods identified through the token itself are looked up (i.e. without the prefixes `get` or `set`). In the previous example a method `add(…)` would be searched for first, and if found, called with the parameters from the body.

For requests on paths with several path segments, it is attempted to execute a Java getter-method for every path token between the root path and the last token. The returned object of a call is wrapped in an invocation wrapper on which in turn methods for the next path token are executed. The call for the final token of the path is then carried out with the previously described call mechanism depending on the HTTP method that is used.

The following examples of possible HTTP requests demonstrate the call mechanism. It is assumed that a `Math` object was registered as remote service under `“/math”`. For illustration, the registered object is referenced by the variable `object` in the examples.

```
// Example of an HTTP request
GET /math/max  HTTP/1.1
```

```
// Excerpt from the class Math
@RestResource (path="math")
```

```
class Math {

      public Math() { }

      @RestResourceMethod

      public int getMax() {
            return Integer.MAX_VALUE;
      }

}
```

The call causes that `object.getMax()` is called and the result of the call (`Integer.MAX_VALUE`) is returned to the caller in the body of the HTTP response url-encoded.

```
// Example of an HTTP request
PUT /math/max  HTTP/1.1

p0=100
```

```
// Excerpt from the class Math
@RestResource (path="math")
class Math {

      public static int max = Integer.MAX_VALUE;

      public Math() {}

      @RestResourceMethod
      public void setMax(int new_max) {
            max=new_max;
      }

}
```

The call causes `object.setMax(100)` to be called and the value of the variable `max` to be reset.

```
// Example of an HTTP request
POST /math/max  HTTP/1.1

p0=100&p1=200
```

```
// Excerpt from the class  Math
@RestResource (path="math")
class Math {

      public static int max = Integer.MAX_VALUE;

      public Math() {}

      @RestResourceMethod
      public int max(int arg1,int arg2) {
            return arg1 > arg2 ? arg1 : arg2;
      }

}
```

The call causes `object.max(100,200)` to be called and the maximum of 100 and 200 (i.e. 200) to be returned to the caller in the body of the HTTP response url-encoded.

To control which methods are exported to remote requests and to be able to set some configurations settings for the resource, Java Annotations are used (see `de.fhg.fokus.restac.resource.wrapping.annotations)`: Each of the exported methods has to be annotated with `@RestResourceMethod`.
Additionally the class itself has to be annotated as a `@RestResource`, then a list of configurations settings can be provided, otherwise default settings are used:

```
* Following parameters can optionally be specified for a resource
* (default values are used in case no specification is given):
*
* protocol - resource accepts only messages following this protocol, for
example http, default is: http
* host - resource accepts messages only from this host (use "" for any
host), default is: any host
* port - resource accepts messages only from this port (use "" for any
port), default is: any port
* path - resource can be accessed by using this path, default is: class
name
* query - resource accepts only this query in a message (URL-encoded query,
use "" for any query), default is: any query
```

5.2 Client-sided functionality of the wrapping model

For the application of the wrapping model, a service allowing a developer to make a Java object transparently available as remote resource, as described in the previous section, is provided through the class `ObjectWrappingFactory` which encapsulates the instantiation of a RESTAC transceiver, as well as the functionality of binding and lookup of methods, to organize the application of the wrapping model as easily and transparently as possible. To achieve this, the object is registered for TCP communication of the given transceiver with the help of a root wrapper.

The method returns a proxy object, which allows the remote call of methods declared in the given interface. The method call is mapped on an HTTP request to the resource with the help of an invocation handler implementation (see Java Reflection API [5]). For this, the peer which manages the addressed resource is discovered by making use of the naming service. If the requested resource was registered on the requesting peer itself, all method calls are executed locally and directly on the wrapped object without any HTTP communication.

For all types which are used for parameters or return values of method calls, a mapping of url-encoding onto the respective Java object type has to be implemented in the class `ParserFunction`, and a mapping of Java object type onto url-encoding in the class `SerializerFunction`.

If the return type of a method represents a Java interface though, a new proxy is being returned. This proxy in turn can be used for further remote calls of methods which are declared in its interface. The accessibility of the resource is verified before the generation of the proxy with the help of a HEAD request. Because of limitations of the Java Reflection API, this process is not possible for types that define a class, and therefore not a Java interface.

The class `ObjectWrappingFactory` provides an interface, which allows developers to easily and transparently make use of the wrapping model in order to make Java objects available as distributed resources or to access remote resources. On instantiation of an object wrapping factory, a new transceiver with the default endpoints for TCP communication is started. Hence, each factory stands for a unique peer. The functionality of an object wrapping factory is illustrated in the following:

`public void registerObject(Object object)`

The method `registerObject()` allows to make Java objects available as distributed resources by registering them under the given name at the server of the factory with a root wrapper and at the naming service.

`public void unregisterObject(String name)`

The method `unregisterObject()` allows to unregister Java objects that were previously registered at this factory and make them not avalaible as distributed resource, anymore.

Examples for the application of the functionality of an object wrapping factory can be found in chapter 2.

References

[1] Roy Thomas Fielding, „Architectural Styles and the Design of Network-based Software Architectures", University of California, Irvine, 2000

[2] IANA MIME Media Types, http://www.iana.org/assignments/media-types/

[3] ISO/IEC 8859-1:1998, "Information technology -- 8-bit single-byte coded graphic character sets Part 1: Latin alphabet No. 1"

[4] RFC 2616, Hypertext Transfer Protocol -- HTTP/1.1, 1999, http://www.w3.org/Protocols/rfc2616/rfc2616.html

[5] Java Reflection API, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/package-summary.html, Java 2 Platform, Standard Edition, v 1.4.2 API Specification

[6] Eclipse Platform API Specification, Release 3.1, http://help.eclipse.org/help31/nftopic/org.eclipse.platform.doc.isv/reference/api/index.html

Appendix A: Configuration

In a small scope, it is possible for the developer to influence the configuration of the RESTAC framework. The file `transceiver.properties` contains a couple of configuration properties, which can be modified by the developer according to his needs. These properties are explained in the following:

```
Outbound[0] = Outbound TCP dispatcher implementation
Inbound[0] = Inbound TCP dispatcher implementation
InboundOutbound[0] = UDP Multicast dispatcher implementation, both inbound
and outbound
InboundOutbound[1] = UDP Unicast dispatcher implementation, both inbound
and outbound
```

Defaults:

```
Outbound[0] =
de.fhg.fokus.restac.httpx.core.dispatcher.client.TCPClientManagedActionMess
ageDispatcher
Inbound[0] =
de.fhg.fokus.restac.httpx.core.dispatcher.server.TCPServerManagedActionMess
ageDispatcher
InboundOutbound[0] =
de.fhg.fokus.restac.httpx.core.dispatcher.UDPMUManagedActionMessageDispatch
er
InboundOutbound[1] =
de.fhg.fokus.restac.httpx.core.dispatcher.UDPUManagedActionMessageDispatche
r
```

Acknowledgements

A first description of a REST-based middleware framework was conducted by Withold Drytkiwicz during his work on his diploma thesis at the Technische Universität Berlin, supervised by Ilja Radusch. Parts of his work were reused for RESTAC. See also following publication:

W. Drytkiwicz, I. Radusch, S. Arbanowski, R. Popescu-Zeletin: pREST: A REST-based Protocol for Pervasive Systems. Proceedings of the 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2004), Fort Lauderdale, USA, October 24-27, 2004, ISBN: 0-7803-8815-1