eProsima RPC over REST

User Manual
Version 0.3.0

The Middleware Experts

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 2 – 1ºG
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Trademarks

eProsima is a trademark of Proyectos y Sistemas de Mantenimiento SL. All other trademarks used in this document are the property of their respective owners.

License

eProsima RPC over REST is licensed under the terms described in the RPCREST_LICENSE file included in this distribution.

Technical Support

- Phone: +34 91 804 34 48
- Email: support@eProsima.com

Contents

1 Introduction

eProsima RPC over REST is a high performance remote procedure call (RPC) framework. It combines a software stack with a code generation engine to build services that will efficiently work in several platforms and programming languages.

REST (Representational State Transfer) is an architectural style consisting of a coordinated set of constraints applied to components, connectors, and data elements, within a distributed hypermedia system. One can characterise RESTful applications by conforming them with the REST principals and constraints.

eProsima RPC over REST supports RESTful as the communication engine to transmit the remote procedure call requests and replies.

1.1 A quick example

You write a .WADL file like this:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://example.com/"/>
        <resource path="Example">
            <method name="GET" id="exampleMethod" />
        </resource>
    </resources>
</application>
```

Then you process the file with the *rpcrestgen* compiler to generate C++ code. Afterwards, you use that code to invoke RESTful resources with the client proxy:

```
ProxyTransport *transport = new HttpProxyTransport(“http://example.com”);
ExampleProtocol *protocol = new ExampleProtocol();
ExampleResourceProxy *proxy = new ExampleResourceProxy(*transport, *protocol);
...
proxy->exampleMethod();
```

or to implement a server using the generated skeleton:

```
HttpServerTransport *transport = new
    HttpServerTransport(“http://example.com”);
ExampleProtocol *protocol = new ExampleProtocol();
SingleThreadStrategy *single = new SingleThreadStrategy();
ExampleResourceServerImpl servant;
ExampleResourceServer *server =
       new ExampleResourceserver(*single, *transport, *protocol, servant);
...
server->serve();
```

See section 4 (HelloWorld example) for a complete step by step example.

1.2 Main features

eProsima RPC over REST provides the developers with a high performance and reliable communication engine (RESTful) to invoke remote procedures. It exposes these features:

- **Synchronous invocation:** The synchronous invocation is the mostly common used approach. It blocks the client's thread until the reply is received from the server.
- **Different threading strategies for the server:** These strategies define how the server acts when a new request is received. The currently supported strategies are:
  - Single-thread strategy: Uses only one thread for every incoming request.
  - Thread-pool strategy: Uses a fixed amount of threads to process the incoming requests.
  - Thread-per-request strategy: Creates a new thread for processing each new incoming request.
- **Complete RESTful Frameworks:** Developers can use RPC over REST code within their applications.

2 Building an application

eProsima RPC over REST allows the developer to easily implement a distributed application using remote procedure invocations. In client/server paradigm, a server offers a set of remote procedures that the client can remotely call. How the client calls these procedures should be transparent.

For the developer, a proxy object represents the remote server, and this object offers the remote procedures implemented by the server. In the same way, how the server obtains a request from the network and how it sends the reply should also be transparent. Hence, the developer just writes the behaviour of the remote procedures.

eProsima RPC over REST offers this transparency and facilitates the development.

The general steps to build an application are:

- Define a set of remote procedures using Interface Definition Language.
- Generate specific code for the remote procedures: a Client Proxy and a Server Skeleton.
- Implement the server: filling the server skeleton with the behaviour of the procedures.
- Implement the client: using the client proxy to invoke the remote procedures.

This section describes the basic concepts of these four steps that the developer has to follow to implement a distributed application. The advanced concepts are described in section *3 (Advanced concepts)*.

2.1 Defining a set of remote procedures

Web Application Description Language (WADL) is used by *eProsima RPC over REST* to define the remote procedures that the RESTful server will offer to its clients. WADL is a machine-readable XML[1] description of HTTP[2]-based web applications (typically REST web services). The WADL complete specification can be consulted by following this link: http://www.w3.org/Submission/wadl/

eProsima RPC over REST includes a Java application named `rpcrestgen`. This application parses the WADL file and transforms it into an IDL (Interface Definition Language) file, generating C++ code afterwards for the specific set of remote procedures that the developer has defined. The `rpcrestgen` application will be described in section 2.3 (*Generating specific remote procedure call support code)*.

WADL is designed to provide a description of every available resource, method, request and response for a RESTful distributed system. RESTful is based on HTTP and uses it as a transport, so the WADL members are HTTP components. We are now going to describe the WADL features that are supported by *eProsima RPC over REST.*

1. Extensible Markup Language (XML) is a language that defines a set of rules for encoding documents in a format that is both human and machine readable. It is based in the XML 1.0 Specification, produced by the W3C (World Wide Web Consortium).
2. HyperText Markup Language (HTML) is the main markup language used for creating web pages and other information that can be displayed in a web browser using hypertext (structured and linked text documents).

Let's see an empty example:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://example.com/resources"/>
</application>
```

Every WADL file has an *application* tag with all the information regarding the remote procedures, and then a tag named *resources* with a base API address. All the resource names are relative to this address.

Let's add a simple resource:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://example.com/resources"/>
        <resource path="customers">
            <method name="GET" id="getCustomers" />
            <method name="POST" id="postCustomer"/>
        </resource>
    </resources>
</application>
```

In the previous image, the resource http://example.com/resources/customers and two HTTP methods (GET and POST) have been defined. We could have PUT and DELETE HTTP methods as well. Note that *id* attributes are optional identifiers for the example.

WADL also supports embedded parameters inside paths, like it is shown in the following example:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://example.com/resources"/>
        <resource path="customers">
            <method name="GET" id="getCustomers" />
            <method name="POST" id="postCustomer"/>
            <resource path="{customerId}">
                <param required="true" type="xsd:int" style="template"
                                                        name="customerId"/>
                <method name="GET" id="getCustomer" />
            </resource>
        </resource>
    </resources>
</application>
```

In this case, we have a new resource http://example.com/resources/customers/{customerId}, where *customerId* is an embedded parameter defined in the following template *param* tag. As the *customerId* type is *int*, a valid URL for this resource would be http://example.com/resources/customers/7. Note that we have another HTTP GET method for this new resource.

When using HTTP you can annex parameters right after the URL, for example: http://example.com/resources/customers?name=John&surname=Doe. In this case, *name* and *surname* are known as *query parameters*.

This kind of parameters are defined this way:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://example.com/resources"/>
        <resource path="customers">
            <method name="GET" id="getCustomers">
                <request>
                    <param name="name" type="xsd:string" style="query"/>
                    <param name="surname" type="xsd:string" style="query"/>
                </request>
            </method>
            <method name="POST" id="postCustomer"/>
            <resource path="{customerId}">
                <param required="true" type="xsd:int" style="template"
                                                        name="customerId"/>
                <method name="GET" id="getCustomer" />
            </resource>
        </resource>
    </resources>
</application>
```

In this version, *eProsima RPC over REST* does not accept optional query parameters.

We have seen two types of parameters so far, but there is still another one to explain. HTTP is usually used to send information inside its body (for example HTML documents), so the RESTful methods support HTTP body parameters via the *representation* tag. Each representation has an *element* attribute. This attribute refers to the element describing the representation format.

For example, if our representation is in XML format, the *element* attribute should be an element description inside an XSD[3] file. WADL can also include these files using the *grammars* tag:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <grammars>
        <include href="CustomerEntry.xsd"/>
    </grammars>
    <resources base="http://example.com/resources"/>
        <resource path="customers">
            ...
            <method name="POST" id="postCustomer">
                <request>
                    <representation id="customerEntry"
                        mediaType="application/xml" element="ce:CustomerEntry"/>
                </request>
            </method>
            ...
        </resource>
    </resources>
</application>
```

We still need to describe the RESTful responses, which are very similar to body parameters, as they both are in the HTTP body. Let's see them with another example:

3. XML Schema Definition (XSD), published as a W3C recommendation, is one of several XML schema languages. It can be used to express a set of rules to which an XML document must conform in order to be considered valid.

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <grammars>
        <include href="CustomerEntry.xsd"/>
        <include href="Responses.xsd"/>
    </grammars>
    <resources base="http://example.com/resources"/>
        <resource path="customers">
            ...
            <method name="POST" id="postCustomer">
                <request>
                    <representation id="customerEntry"
                        mediaType="application/xml" element="ce:CustomerEntry"/>
                </request>
                <response status="200">
                    <representation id="resultSet"
                        mediaType="application/xml" element="rs:ResultSet"/>
                </response>
                <response status="400">
                    <representation id="error"
                        mediaType="application/xml" element="rs:Error"/>
                </response>
            </method>
            ...
        </resource>
    </resources>
</application>
```

In this HTTP POST method, we have two possible responses: one with a *ResultSet* element and another with an *Error* element. Both in XML format.

The current version of *eProsima RPC over REST* supports both XML and JSON[4] formats for body parameters and responses, but dealing with them as strings. Parsing these strings is the user's responsibility, so *RPC over REST* will ignore the *include* tags.

One more feature that WADL supports is to declare specific tags as references to the global ones. We do not have to declare several times the same *method*, *representation* or *param* that we use in multiple resources.

For this purpose we can use the *href* attribute as it is used in this example.

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    ...
    <resources base="http://example.com/resources"/>
        <resource path="customers">
            ...
            <method href="#postCustomer" />
            ...
        </resource>
    </resources>

    <method name="POST" id="postCustomer">
        <request>
            <representation href="#customerEntry" />
        </request>
        ...
    </method>
```

4. JSON or JavaScript Object Notation is an open source standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs.

```
    <representation id="customerEntry" mediaType="application/xml"
                                        element="ce:CustomerEntry"/>
</application>
```

2.2 WADL syntax and mapping to IDL and C++

Embedded URL (Uniform Resource Locator) parameters and query parameters must not be complex types, since they are part of the URL, and they must have an exact representation as a string.

Body parameters and requests can be complex types, but we are going to treat them as their string representation, so RPC over REST does not support any kind of complex type in the current version.

2.2.1 Simple parameter types

eProsima RPC over REST supports a variety of simple types that the developer can use in the method query and embedded parameters. The following table shows the supported simple types, how they are translated into IDL and what the `rpcrestgen` application generates in C++ language.

TABLE 1: SPECIFYING SIMPLE TYPES IN WADL FOR C++

WADL type	IDL type	Sample C++ Output Generated by rpcrestgen
xsd:string	`string`	`Std::string string_member` `/* maximum length = (255) */`
xsd:byte	`char`	`char char_member`
xsd:unsignedByte	`octet`	`uint8_t octet_member`
xsd:short	`short`	`int16_t short_member`
xsd:unsignedShort	`unsigned short`	`uint16_t ushort_member`
xsd:int	`long`	`int32_t long_member`
xsd:unsignedInt	`unsigned long`	`uint32_t ulong_member`
xsd:long	`long long`	`int64_t llong_member`
xsd:unsignedLong	`unsigned long long`	`uint64_t ullong_member`
xsd:float	`float`	`float float_member`
xsd:double	`double`	`double double_member`
xsd:boolean	`boolean`	`bool boolean_member`

2.2.2 Parameter definition

As we saw in section 2.1 (Defining a set of remote procedures), the supported *param* tags can be children either of a *resource* or a *request*. These are the supported attributes and values:

Attribute	Meaning
id	Optional. May be used to refer to this parameter elsewhere through *href* attribute.
name	Required name of the parameter.
style	Parameter style. Must be *template* for embedded parameters and *query* for query parameters.
type	Parameter type. Supported types have been seen in the previous subsection.
href	Used for *parameter* references. Refers to a global *parameter id* attribute.

In the IDL file, each WADL *query* parameter will become a function parameter, and every *template* parameter will be part of a structure containing all the embedded parameters.

This structure will finally be the first parameter of every method which is affected by these embedded parameters mentioned before.

2.2.3 Representation definition

A *representation* may be a body parameter or a response, and it must be a child of a *request* or a *response* tag. Its attributes are:

Attribute	Meaning
id	Optional. May be used to refer to this *representation* elsewhere through *href* attribute.
mediaType	Media type of the representation. Supported types are *application/xml* and *application/json*
element	Qualified name of the root element as described in the *grammars* tag.
href	Used for *representation* references. Refers to another *representation id* attribute.

In the IDL file, *representation* tags are translated depending on their parents.

A *request* representation can have several formats. We support both XML and JSON formats, so request representations will become the members inside an IDL union. A union is a type that specifies which of a number of permitted types may be stored in its instances.

The generated IDL union for a Post method will be:

```
union PostRequest switch(long)
{
    case 1:
        string xmlRepresentation;
    case 2:
        string jsonRepresentation;
};
```

If a method has a request representation, this union will be its last parameter.

When this union is translated to C++, a class is generated. It has three members: a discriminant named *_d* and all the union possibilities. The member in use is indicated by the discriminant. It would be like it is shown in the following example:

```
class PostRequest {
public:
    /** Constructors **/
    PostRequest();
    ...

    /** Discriminator **/
    int32_t _d();
    void _d(int32_t x);
    ...

    /** Getter and Setters **/
    std::string xmlRepresentation();
    std::string jsonRepresentation();
    ...

private:
    int32_t m__d;
    std::string m_xmlRepresentation; /* maximum length = (255) */
    std::string m_jsonRepresentation; /* maximum length = (255) */
};
```

A *response* representation will be a little bit different. We support XML and JSON responses, but it could also be empty, and every response has an HTTP status code.

An IDL response representation for a method named *post* could be this set of unions and structures:

```
struct EmptyPostResponse
{
    long status;
};

struct XMLPostResponse
{
    long status;
    string xmlRepresentation;
};

struct JSONPostResponse
{
    long status;
    string jsonRepresentation;
};

union PostResponse switch(long)
{
    case 0:
        EmptyPostResponse emptyPostResponse;
    case 1:
        XMLPostResponse xmlPostResponse;
    case 1:
        JSONPostResponse jsonPostResponse;
};
```

When this set of unions and structures is translated to C++, a set of classes are generated. The generated classes are shown in the following example:

```
class EmptyPostResponse
{
public:
    /** Constructors **/
    EmptyPostResponse();
    ...

    /** Getters and Setters **/
    int32_t status();
    void status(int32_t x);
    ...

private:
    int32_t m_status;
};

class XMLPostResponse
{
public:
    /** Constructors **/
    XMLPostResponse();
    ...
```

```
    /** Getters and Setters **/
    int32_t status();
    std::string xmlRepresentation();
    ...

private:
    int32_t m_status;
    std::string m_xmlRepresentation; /* maximum length = (255) */
};

class JSONPostResponse
{
public:
    /** Constructors **/
    JSONPostResponse();
    ...

    /** Getters and Setters **/
    int32_t status();
    std::string jsonRepresentation();
    ...

private:
    int32_t m_status;
    std::string m_jsonRepresentation; /* maximum length = (255) */
};

class PostResponse {
public:
    /** Constructors **/
    PostResponse();
    ...

    /** Discriminator **/
    int32_t _d();
    void _d(int32_t x);
    ...

    /** Getter and Setters **/
    EmptyPostResponse  emptyPostResponse();
    XMLPostResponse xmlPostResponse();
    JSONPostResponse jsonPostResponse();
    ...

private:
    int32_t m__d;
    EmptyPostResponse m_emptyPostResponse;
    XMLPostResponse m_xmlPostResponse;
    JSONPostResponse m_jsonPostResponse;
};
```

2.2.4 Method definition

This tag describes the input and output from an HTTP protocol method that may be applied to a resource. Its attributes are:

Attribute	Meaning
name	HTTP method. Can be *GET*, *POST*, *PUT* or *DELETE*.
id	Optional. May be used to refer to this *method* elsewhere through *href* attribute.
href	Used for *method* references. Refers to another *method id* attribute.

A *method* tag has the following child elements:

- A *request* tag describing the input as a collection of *param* tags and an optional *representation*.
- Zero or more *response* tags describing the possible *method* outputs.

Note that WADL methods inherit embedded URL parameters from their parent resources.

In the IDL file, the method will become a function, and its parameters and response types have already been explained with *param* and *response* tag examples.

2.2.5 Resource definition

A resource tag describes a set of resources, each one identified by a URI. It could have the following attribute:

Attribute	Meaning
path	Provides a URI relative to its parent URI..

A *resource* tag might have zero or more child *resource* tags. Each child resource inherits its parent *path* and its *template* parameters.

On the IDL file, every *resource* will become an *interface*.

2.2.6 Resources definition

The *resources* tag acts as a container for the resources provided by the application. This element has one attribute:

Attribute	Meaning
base	Base URI for the application.

This tag may have zero or more *resource* tags, which inherit the base URI.

2.2.7 Application definition

The *application* tag contains the whole API for the distributed system. It has a *resources* child tag and zero or more global *param*, *representation* or *resource* tags.

2.2.8 Example

WADL syntax described in this section is shown through an example. This example is called *Bank.wadl* and its content is:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://example.com/resources/">
        <resource path="account/{accountNumber}">
            <param name="accountNumber" type="xsd:int" style="template" />
            <method name="POST" id="getAccountDetails">
                <request>
                    <representation href="#password" />
                    <param href="#user" />
                </request>
                <response status="200">
                    <representation href="#resultSet" />
                </response>
                <response status="400">
                    <representation href="#errorMessage" />
                </response>
            </method>
        </resource>
    </resources>
<param id="user" name="user" type="xsd:string" style="query" />
    <representation id="password" mediaType="application/xml"
        element="Password" />
    <representation id="resultSet" mediaType="application/xml"
        element="AccountDetails" />
    <representation id="errorMessage" mediaType="application/xml"
        element="ErrorMessage" />
</application>
```

This example will be used as a base to other examples in the following sections.

2.3 Generating specific remote procedure call support code

Once the API is defined in a WADL file, we need to generate the code for a client proxy and a server. *eProsima RPC over REST* provides the `rpcrestgen` tool for this purpose. This tool parses the WADL file, converting it into an IDL file and generating the corresponding supporting C++ code.

2.3.1 RPCRESTGEN command syntax

The general syntax is:

```
rpcrestgen [options] <WADL file> <WADL file> ...
```

Where the options could be:

Option	Description
-help	Shows help information
-version	Shows the current version of *eProsima RPC over REST*
-ppPath <directory>	Location of the C/C++ preprocessor.
-ppDisable	Disables the C/C++ preprocessor. Useful when macros or includes are not used.
-example <platform>	Creates a solution for a specific platform. This solution

	will be used by the developer to compile both client and server. **Possible values:** i86Win32VS2010, x64Win64VS2010, i86Linux2.6gcc4.4.5, x64Linux2.6gcc4.4.5
-replace	Replaces existing generated files.
-d <path>	Sets an output directory for generated files
-t <temp dir>	Sets a specific directory as a temporary directory

The rpcrestgen application generates several files that will be described in this section. Their names are generated using the WADL file name. The <*WADLName*> tag has to be substituted by the WADL file name.

2.3.2 Server side

rpcrestgen generates C++ source files with the definitions of the remote procedures and C++ header files with the declaration of these remote procedures. These files are the skeletons of the servants that implement the defined resources. The developer can use each definition in the source files to implement the behaviour of the remote procedures. These files are <WADL*Name*>ServerImpl.h and <WADL*Name*>ServerImpl.cxx. rpcrestgen also generates a C++ source file with an example of a server application and a server instance. This file is <*WADLName*>ServerExample.cxx.

2.3.3 Client side

rpcrestgen generates a C++ source file with an example of a client application and how this client application can call a remote procedure from the server. This file is <*WADLName*>ClientExample.cxx.

2.4 Server implementation

After the execution of rpcrestgen, two files named <WADL*Name*>ServerImpl.cxx and <WADL*Name*>ServerImpl.h will be generated. These files are the skeleton of the resources offered by the server. All the remote procedures are defined in these files, and the behaviour of each one has to be implemented by the developer. For the remote procedure *getAccountDetails* seen in our example, the generated definition is:

```
GetAccountDetailsResponse
account_accountNumberResourceServerImplExample::getAccountDetails(
    /*in*/ const account_accountNumber& account_accountNumber,
    /*in*/ const std::string& user,
    /*in*/ const GetAccountDetailsRequest& GetAccountDetailsRequest)
{
    GetAccountDetailsResponse getAccountDetails_ret;

    return getAccountDetails_ret;
}
```

The code generated by rpcrestgen also contains the server classes. These classes are implemented in the files <WADL*Name*>Server.h and <WADL*Name*>Server.cxx. They offer the resources implemented by the servants. When an object of these classes defined in <WADL*Name*>Server.h is created, a connection can be established between the proxy and the server.

2.4.1 API

Using the suggested WADL example, the API created for this class is:

```
class account_accountNumberResourceServer : public eprosima::rpc::server::Server
{
public:
account_accountNumberResourceServer(
            eprosima::rpc::strategy::ServerStrategy &strategy,
            eprosima::rpc::transport::ServerTransport &transport,
            eprosima::rpc::protocol::BankProtocol &protocol,
            account_accountNumberResourceServerImpl &servant
        );

    virtual ~account_accountNumberResourceServer();

private:
    account_accountNumberResourceServerImpl &_impl;
};
```

The server provides a constructor with four parameters. The `strategy` parameter expects a server's strategy that defines how the server has to manage incoming requests. Server strategies are described in section 3.2 (Threading server strategies).

The second parameter expects the network transport that will be used to establish the connections with the proxies. For RESTful applications, it will be an *HTTPServerTransport*. The third parameter is the protocol. It's generated by *rpcrestgen* and it's the class that deserializes received data and gives it to the user implementation. Finally, the fourth parameter is the server skeleton implemented by the user, for example by filling the empty example given.

2.4.2 Example

Using the suggested WADL example, the developer can create a server in the following way:

```
unsigned int threadPoolSize = 5;
ThreadPoolStrategy *pool = NULL;
BankProtocol *protocol = NULL;
HTTPServerTransport *transport = NULL;
account_accountNumberResourceServer *server = NULL;
account_accountNumberResourceServerImplExample servant;

try
{
    pool = new ThreadPoolStrategy(threadPoolSize);
    protocol = new BankProtocol();
    transport = new HTTPServerTransport("192.168.1.14:8080");
    server = new account_accountNumberResourceServer(*pool, *transport,
                                                     *protocol, servant);
    server->serve();
}
catch(InitializeException &ex)
{
    std::cout << ex.what() << std::endl;
    return -1;
}
```

2.5 Client implementation

The code generated by `rpcrestgen` contains classes that act like proxies of the remote servers. These classes are implemented in the files `<WADL*Name*>Proxy.h` and `<WADL*Name*>Proxy.cxx`. The proxies offer the server resources and the developer can directly invoke its remote procedure.

2.5.1 API

Using the suggested WADL example, the API of this class is:

```
class account_accountNumberResourceProxy : public eprosima::rpc::proxy::Proxy
{
public:
    account_accountNumberResourceProxy(
            eprosima::rpc::transport::ProxyTransport &transport,
            eprosima::rpc::protocol::BankProtocol &protocol);

    virtual ~account_accountNumberResourceProxy();

    GetAccountDetailsResponse getAccountDetails(
            /*in*/ const account_accountNumber& account_accountNumber,
            /*in*/ const std::string& user,
            /*in*/ const GetAccountDetailsRequest& GetAccountDetailsRequest);
};
```

The proxy provides a constructor. It expects the network transport that will be used to establish the connection with the server as a parameter. It must be an instance of *HTTPProxyTransport*. The second parameter is the protocol. Again, it is generated by `rpcrestgen` and its duty is to serialize and deserialize protocol data.

The proxy provides the remote procedures to the developer. Using the suggested WADL, our proxy will provide the remote procedure `getAccountDetails`.

2.5.2 Example

By using the suggested WADL example, the developer can invoke *`getAccountDetails`* procedure in the following way:

```
BankProtocol *protocol = NULL;
ProxyTransport *transport = NULL;
account_accountNumberResourceProxy *proxy = NULL;

// Creation of the proxy for interface "account_accountNumberResource".
try
{
    protocol = new BankProtocol();
    transport = new HTTPProxyTransport("127.0.0.1:8080");
    proxy = new account_accountNumberResourceProxy(*transport, *protocol);
}
catch(InitializeException &ex)
{
    std::cout << ex.what() << std::endl;
    return -1;
}

// Create and initialize parameters.
account_accountNumber account_accountNumber;
```

```
std::string user;
GetAccountDetailsRequest GetAccountDetailsRequest;

// Create and initialize return value.
GetAccountDetailsResponse getAccountDetails_ret;

// Call to remote procedure "getAccountDetails".
try
{
    getAccountDetails_ret =
        proxy->getAccountDetails(account_accountNumber,
                                 user, GetAccountDetailsRequest);
}
catch(SystemException &ex)
{
    std::cout << ex.what() << std::endl;
}
```

3 Advanced concepts

3.1 Network transports

eProsima RPC over REST provides several network transports, but RESTful needs HTTP to run.

3.1.1 HTTP Transport

The purpose of this transport is to create a keep-alive TCP connection between a proxy and a server that will communicate through HTTP. This transport is implemented by two classes. One is used by server proxies and the other one is used by servers.

HttpProxyTransport:

`HttpProxyTransport` class implements an HTTP transport that must be used by proxy servers.

```
class HttpProxyTransport : public ProxyTransport
{
public:
    HttpProxyTransport(const std::string &serverAddress);
    virtual ~HttpProxyTransport();
    ...
private:
    TCPProxyTransport m_tcptransport;
    ...
};
```

This class has a constructor with a parameter that receives the server URL to connect to.

Using our suggested WADL example, the developer can create a proxy that connects to a specific server:

```
BankProtocol *protocol = NULL;
ProxyTransport *transport = NULL;
account_accountNumberResourceProxy *proxy = NULL;

// Creation of the proxy for interface "account_accountNumberResource".
try
{
    protocol = new BankProtocol();
    transport = new HttpProxyTransport("192.168.1.14:8080");
    proxy = new account_accountNumberResourceProxy(*transport,
                                                   *protocol);
}
catch(InitializeException &ex)
{
    std::cout << ex.what() << std::endl;
    return -1;
}
```

HttpServerTransport:

`HttpServerTransport` class implements an HTTP transport that must be used by servers.

```
class HttpServerTransport : public ServerTransport
{
public:
    HttpServerTransport(const std::string &to_connect);
    virtual ~HttpServerTransport();
    ...
private:
    TCPServerTransport m_tcptransport;
    ...
};
```

This class has a constructor which receives a parameter representing the IP address and port that the server will be using to read incoming requests.

By using our suggested WADL example, the developer can create a server that will wait for proxy requests this way:

```
unsigned int threadPoolSize = 5;
ThreadPoolStrategy *pool = NULL;
BankProtocol *protocol = NULL;
HttpServerTransport *transport = NULL;
account_accountNumberResourceServer *server = NULL;
account_accountNumberResourceServerImplExample servant;

try
{
    pool = new ThreadPoolStrategy(threadPoolSize);
    protocol = new BankProtocol();
    transport = new HttpServerTransport("192.168.1.14:8080");
    server = new account_accountNumberResourceServer(*pool, *transport,
                                                     *protocol, servant);
    server->serve();
}
catch(InitializeException &ex)
{
    std::cout << ex.what() << std::endl;
    return -1;
}
```

3.2 Threading server strategies

RPC over REST library offers several strategies that the server may use when a request arrives. This subsection describes these strategies.

3.2.1 Single thread strategy

This is the simplest strategy, in which the server only uses one thread for doing the request management. In this case the server only executes one request at the same time. The thread used by the server to handle the request is the REST reception thread. To use *Single Thread Strategy*, create the server providing the constructor with a `SingleThreadStrategy` object.

```
SingleThreadStrategy * single = NULL;
BankProtocol *protocol = NULL;
HTTPServerTransport *transport = NULL;
account_accountNumberResourceServer *server = NULL;
account_accountNumberResourceServerImplExample servant;

try
{
    single = new SingleThreadStrategy();
    protocol = new BankProtocol();
    transport = new HTTPServerTransport("192.168.1.14:8080");
    server = new account_accountNumberResourceServer(*single, *transport,
                                                     *protocol, servant);
    server->serve();
}
catch(InitializeException &ex)
{
    std::cout << ex.what() << std::endl;
    return -1;
}
```

3.2.2 Thread Pool strategy

In this case, the server manages a thread pool that will be used to process the incoming requests. Every time a request arrives, the server assigns it to a free thread located in the thread pool.

To use *Thread Pool Strategy*, create the server providing the constructor with a `ThreadPoolStrategy` object.

```
unsigned int threadPoolSize = 5;
ThreadPoolStrategy *pool = NULL;
BankProtocol *protocol = NULL;
HTTPServerTransport *transport = NULL;
account_accountNumberResourceServer *server = NULL;
account_accountNumberResourceServerImplExample servant;

try
{
    pool = new ThreadPoolStrategy(threadPoolSize);
    protocol = new BankProtocol();
    transport = new HTTPServerTransport("192.168.1.14:8080");
    server = new account_accountNumberResourceServer(*pool, *transport,
                                                     *protocol, servant);
    server->serve();
}
catch(InitializeException &ex)
{
    std::cout << ex.what() << std::endl;
    return -1;
}
```

3.2.3 Thread per request strategy

In this case, the server will create a new thread for each new incoming request.

To use the Thread per request Strategy, create the server providing it with a `ThreadPerRequestStrategy` object in the constructor method.

```
ThreadPerRequestStrategy * perRequest = NULL;
BankProtocol *protocol = NULL;
HTTPServerTransport *transport = NULL;
account_accountNumberResourceServer *server = NULL;
account_accountNumberResourceServerImplExample servant;

try
{
    perRequest = new ThreadPerRequestStrategy();
    protocol = new BankProtocol();
    transport = new HTTPServerTransport("192.168.1.14:8080");
    server = new account_accountNumberResourceServer(*perRequest,
                                                     *transport,
                                                     *protocol, servant);
    server->serve();
}
catch(InitializeException &ex)
{
    std::cout << ex.what() << std::endl;
    return -1;
}
```

4 Hello World example

In this section an example on how to use this library is explained step by step. In this example, only one remote procedure is defined. A client can invoke this remote procedure by passing a string with a name as a parameter. The server returns a new string that appends the name to a greeting sentence.

4.1 Writing the WADL file

Define a simple resource named `HelloWorld` with a method named `hello`. Store this WADL definition in a file named `HelloWorld.wadl`:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://example.com/resources/">
        <resource path="HelloWorld">
            <method name="GET" id="hello">
                <request>
                    <param name="name" type="xsd:string" style="query"/>
                </request>
                <response status="200">
                    <representation id="helloWorldResponse"
                            mediaType="application/xml" element="Response" />
                </response>
            </method>
        </resource>
    </resources>
</application>
```

4.2 Generating specific code

Open a command prompt and go to the directory containing `HelloWorld.wadl` file. If you are running this example in Windows, type in and execute the following line:

```
rpcrestgen -example x64Win64VS2010 HelloWorld.wadl
```

If you are running it in Linux, execute this one:

```
rpcrestgen -example x64Linux2.6gcc4.4.5 HelloWorld.wadl
```

Note that if you are running this example in a 32-bit operating system you have to use *-example i86Win32VS2010* (*i86Linux2.6gcc4.4.5* in Linux) instead.

This command translates the WADL file into an IDL file. It also generates the client stub and the server skeletons, as well as some project files designed to build your HelloWorld example.

In Windows, a Visual Studio 2010 solution will be generated, named *rpcsolution-<target>.sln*, being *<target>* the chosen example platform. This solution is composed by five projects:

- *HelloWorld*, with the common classes of the client and the server, like the defined types and the specific communication protocol

- *HelloWorldServer*, with the server code

- *HelloWorldClient*, with the client code.

- HelloWorldServerExample, with a usage example of the server, and the implementation skeleton of the RPCs.

- HelloWorldClientExample, with a usage example of the client

In Linux, on the other hand, it generates a makefile with all the required information to compile the solution.

4.3 Client implementation

Edit the file named `HelloWorldClientExample.cxx`. In this file, the code for invoking the *hello* RPC using the generated proxy will be generated. You have to add two more statements: one to set a value to the remote procedure parameter and another to print the returned value. This is shown in the following example:

```
int main(int argc, char **argv)
{
    HelloWorldProtocol *protocol = NULL;
    ProxyTransport *transport = NULL;
    HelloWorldResourceProxy *proxy = NULL;

    // Creation of the proxy for interface "HelloWorldResource".
    try
    {
        protocol = new HelloWorldProtocol();
        transport = new HTTPProxyTransport("127.0.0.1:8080");
        proxy = new HelloWorldResourceProxy(*transport, *protocol);
    }
    catch(InitializeException &ex)
    {
        std::cout << ex.what() << std::endl;
        return -1;
    }

    // Create and initialize parameters.
    std::string name = "Richard"; // Set the remote procedure parameter

    // Create and initialize return value.
    HelloResponse hello_ret;

    // Call to remote procedure "hello".
    try
    {
        hello_ret = proxy->hello(name);

        if(hello_ret._d() == 1) {
            std::cout << "HTTP Status: " <<
                hello_ret.xmlHelloResponse().status() << std::endl;
            std::cout << "HTTP Response: " <<
                hello_ret.xmlHelloResponse().xmlRepresentation() << std::endl;
        }
    }
    catch(SystemException &ex)
    {
        std::cout << ex.what() << std::endl;
    }
```

```
    delete(proxy);
    delete(transport);
    delete(protocol);

    return 0;
}
```

4.4 Server implementation

`rpcrestgen` creates the server skeleton in the file `HelloWorldServerImplExample.cxx`. The remote procedure is defined in this file and it has to be implemented.

In this example, the procedure returns a new string appended to a greeting sentence. Open the file and copy this code for implementing such behaviour:

```
#include "HelloWorldServerImplExample.h"

HelloResponse HelloWorldResourceServerImplExample::hello(/*in*/ std::string &
name)
{
    HelloResponse hello_ret;

    hello_ret._d() = 1; // 1 -> XML representation
    hello_ret.xmlHelloResponse().status(200); // 200 -> HTTP OK
    hello_ret.xmlHelloResponse().xmlRepresentation("<Response>Hello “ + name +
“!</Response>");

    return hello_ret;
}
```

It's important for the developer to know how to fill the *<Method>Response* union, being *<Method>* the name of out method. We support empty responses, XML responses and JSON responses. First, we have to fill the discriminator, *_d*. We will use 0 for empty responses, 1 for XML and 2 for JSON. Then, depending on our discriminator, we will fill one of the following members:

- `<Method>Response.empty<Method>Response`
- `<Method>Response.xml<Method>Response`
- `<Method>Response.json<Method>Response`

All of them are structures, and they have a numeric *status* member, where the developer must write the HTTP status code. `xml<Method>Response` also has a std::string data type called *xmlRepresentation*, while in `json<Method>Response` we can find a std::string named *jsonRepresentation*.

4.5 Build and execute

To build your code using Visual Studio 2010, make sure you are in the Debug (or Release) profile, and then build it (F7). Now go to *<example_dir>*`\bin\x64Win64VS2010` directory and execute `HelloWorldServerExample.exe`. You will get the message:

```
INFO<eprosima::rpc::server::Server::server>: Server is running
```

Then launch `HelloWorldClientExample.exe`. You will see the result of the remote procedure call:

```
HTTP Status: 200
HTTP Response: <Response>Hello Richard!</Response>
```

This example was created statically. To create a set of DLLs containing the protocol and the structures, select the Debug DLL (or Release DLL) profile and build it (F7). Now, to get your DLL and LIB files, go to `<example_dir>\objs\x64Win64VS2010` directory. You can now run the same application dynamically using the .exe files generated in `<example_dir>\bin\x64Win64VS2010`, but first you have to make sure your .dll location directory is appended to the PATH environment variable.

To build your code in Linux use this command:

```
make -f makefile_x64Linux2.6gcc4.4.5
```

No go to `<example_dir>\bin\x64Linux2.6gcc4.4.5` directory and execute the binaries as it has been described for Windows.