RPCDDS – Restful support – DETAILED DESIGN

|  |  |
| --- | --- |
| DOCUMENT INFORMATION | |
| Name | RPCDDS – Restful support – Detailed design |
| Product | Documentation |
| Author | Ricardo González |
| Distribution | Internal |
| Version | 1.0 |
| Date | 3rd January 2013 |
| Status | Draft |
| Contact | ricardogonzalez@eprosima.com |
| Document Code |  |

Content

[1 Introduction 3](#_Toc338259247)

[1.1 Target 3](#_Toc338259248)

[1.2 Audience 3](#_Toc338259249)

[1.3 Related documentation 3](#_Toc338259250)

[1.4 Definitions and acronyms 3](#_Toc338259251)

[2 Evaluation 4](#_Toc338259252)

[3 Static structure model 6](#_Toc338259253)

[3.1 Study of WADL 7](#_Toc338259254)

[3.1.1 Mapping WADL to RPCDDS IDL 7](#_Toc338259255)

[3.1.2 How WADL maps remote procedures into HTTP messages 9](#_Toc338259256)

[3.2 Protocol classes 9](#_Toc338259257)

[3.2.1 Hiding the type of the serialized data 10](#_Toc338259258)

[3.2.2 Detecting the remote procedure 12](#_Toc338259259)

[3.3 Transport classes 12](#_Toc338259260)

[3.3.1 Restful TCP transport 13](#_Toc338259261)

[3.4 Server’s proxy 13](#_Toc338259262)

[3.5 Server 15](#_Toc338259263)

[4 Behavior model 16](#_Toc338259264)

[5 Unit test design 17](#_Toc338259265)

# Introduction

## Target

The purpose of this document is to design a new feature in RPCDDS. Restful architectural style will be supported in RPCDDS. This means that a client application developed with RPCDDS could call remote procedures of a remote server that provides a Restful interface. Also a server application developed with RPCDDS could provide a Restful interface to client applications.

## Audience

This document has been written for the developers of this project.

## Related documentation

* [Web Application Description Language Submission](http://www.w3.org/Submission/wadl/)

## Definitions and acronyms

**WADL:** Web Application Description Language

# Evaluation

The new implementation must implement next features to achieve complete Restful support.

* WADL support. WADL is used to describe Restful web services. Considering that Restful wants to be supported in RPCDDS, a user should be able to describe its remote procedure calls in this description language. Then there must be a mapping between the RPCDDS IDL format and the WADL format. The goal of this mapping is to create an application that is capable of transform WADL file to a RPCDDS IDL file and vice versa.
* Future RPCDDS API for users has to coexist with current RPCDDS API and it will increase its functionality. Some example of this new functionality would be: user could connect a remote server using Restful or user could create a server that supports Restful and RPC using DDS.
* RPCDDS should provide an internal API to serialize and deserialize Restful data. This internal API comes out from the study of how WADL represents its content in the HTTP protocol and the study of mapping WADL and RPCDDS IDL.
* RPCDDS code generator must generate specific code to serialize and deserialize the remote procedures defined by user using RPCDDS IDL. For this purpose this specific code will use the internal API to serialize and deserialize Restful data.
* RPCDDS will implement one or more transports that allow sending and receiving HTTP packages. The main transport should be a TCP transport.
* RPCDDS should implement a mechanism to know the source of a request, whether it comes from DDS or a Restful transport. This knowledge is useful to know where the reply has to be sent, using DDS or the Restful transport.

This section will evaluate the coexistence between the current RPCDDS implementation and the new features’ implementation. First step is to review the current architecture of RPCDDS.

**DDS**

ClientRPC

ClientRPC

ClientRPC

ServerRPC

ServerRPC

ServerRPC

**Client**

**Server**

RPCDDS: Current architecture

When a user implements a client application, its application interacts with an object that acts like a server’s proxy. This object exposes the set of remote procedures of the remote server. Common functionality of this object is implemented by the class eProsima::DDSRPC::Client. Also this class manages the DDS Domain Participant entity. For each remote procedure, the class eProsima::DDSRPC::Client manages an object. This object manages the call of the remote procedure that a user makes and common functionality of this object is implemented by the class eProsima::DDSRPC::ClientRPC. Also this class manages the DDS DataWriter entity which sends the request and the DDS Datareader entity which receives the reply.

When a user implements a server application, user has to implement a generated skeleton. This skeleton is managed by an object that acts like the server and the common functionality of this object is implemented by the class eProsima::DDSRPC::Server. This class manages the DDS Domain Participant. For each remote procedure, the class eProsima::DDSRPC::Server manages an object. This object manages the reception of requests from a remote procedure that a client makes and common functionality of this object is implemented by the class eProsima::DDSRPC::ServerRPC*.* Also this class manages the DDS Datareader entity which receives the request and the DDS Datawriter entity which sends the reply.

Having seen the current architecture, next step is to find out the common points that current implementation and new features’ implementation will have in the future implementation.

At user’s level, the interaction has to be the same. In the case of a client application, the server’s proxy has to exposes the set of remote procedures in the same way. Also in the case of a server application, the skeleton has to be generated in the same way.

After a client application calls a remote procedure using the server’s proxy, it could be seen common steps in the way a remote procedure is carried out using DDS and using Restful.

First it will be analyzed the way using DDS. When a client application calls a remote procedure using the server’s proxy, the remote procedure and its input parameters are encapsulated in a DDS type. You could say that there was a serialization of the remote procedure and its input parameters. Then the DDS type is sent using DDS. DDS is acting as a transport. When DDS receives a reply, this reply is a DDS type. Output parameters of the remote procedure were get deserializing the DDS type and then they are returned to the client application.

Now when a client application wants to use Restful, the input parameters of the remote procedure will be encapsulated in a HTTP message. Then this HTTP message will be sent using a transport that works with a socket in its low-level. When the transport receives a reply, this reply is an HTTP message. Output parameters of the remote procedure get deserializing the HTTP message and then they are returned to the client application.

In the design of new RPCDDS implementation it will be taken into account that there are two common points: both mechanisms use a protocol to serialize and deserialize, and both use a way to send and receive the data.

# Static structure model

The proposed model in this document tries to give an RPC framework for the user. The internal use of subscriber/publisher architecture or Restful philosophy must to be transparent. The user only wants to execute remote procedures and obtain the results. This document will be focused in the Restful philosophy and its integration in the RPCDDS product.

As Restful philosophy wants to be used, the concept of remote procedure call is mapped in sending a request from client to the server and receiving a reply in the other way. Both, request and reply, are HTTP messages. How RPCDDS will map remote procedures in a HTTP messages is defined by WADL and how it maps a remote procedure in a HTTP message. The section 3.1 makes a study of WADL. The subsection 3.1.1 studies how WADL maps its definitions in HTTP methods. This will help in the creation of an internal API in the section (TODO: insert section). This API will be use to serialize definitions in RPCDDS IDL into HTTP messages. The subsection 3.1.2 studies how to convert a WADL file into a RPCDDS IDL file.

As it was commented, one common point for DDS and Restful is the serialization. Both mechanisms have to serialize the remote procedure and its parameters. Section 3.2 explains the protocol classes that must be generated and they are responsible for serializing the remote procedure and its parameters.

Whether DDS middleware is seen like a transport, the other common point is the transport level. There will be several transports. One of them encapsulates the usage of DDS middleware to send the DDS data, and the others are used to send Restful protocol data. Section 3.3 explains the transport classes.

IDL compiler continues to be an important part in the design. From now on IDL compiler not only generates code to serialize into DDS types, but it will generate code to serialize into HTTP messages. Section (TODO: insert section) talks about the changes in the IDL compiler.

In the rest of this document when it appears a generated class or generated code by the IDL compiler, the next RPCDDS IDL example will be use:

// file Bank.idl

enum ReturnCode

{

SYSTEM\_ERROR,

ACCOUNT\_NOT\_FOUND,

AUTHORIZATION\_ERROR,

NOT\_MONEY\_ENOUGH,

OPERATION\_SUCCESS

};

struct Account

{

string AccountNumber;

string Username;

string Password;

}; //@top-level false

interface Bank

{

ReturnCode deposit(in Account ac, in long money);

};

## Study of WADL

WADL is designed to provide a machine process-able description of HTTP-based Web applications. WADL describes the following aspects of a Web application:

* *Set of resources.* Analogous to a site map showing the resources on offer.
* *Relationships between resources.* Describing the links between resources, both referential and causal.
* *Methods that can be applied to each resource.* The HTTP methods that can be applied to each resource, the expected inputs and outputs and their supported formats.
* *Resource representation formats.* The supported MIME types and data schemas in use.

RPCDDS IDL should be able to describe these aspects too. For this purpose this study will be map the WADL description components to RPCDDS IDL and some new improvements should be added to RPCDDS IDL to support some features.

### Mapping WADL to RPCDDS IDL

This section describes the WADL description components and how they could be mapped in RPCDDS IDL. All WADL elements have the following XML namespace name:

<http://wadl.dev.java.net/2009/02>

#### Cross Referencing

#TODO

#### Application

The application element forms the root of a WADL description and contains the following:

1. Zero or more doc elements - see section
2. An optional grammars element - see section
3. Zero or more resources elements - see section
4. Zero or more of the following:
   * resource\_type elements - see section
   * method elements - see section
   * representation elements - see section
   * param elements - see section

This element is not needed to be mapped to RPCDDS IDL. The main IDL file will contain all elements that application element contains.

In the case that a target namespace is defined, all elements that the application element contains will be contained inside an IDL module element whose name will be gotten from the target namespace. XML target namespace are defined in the root of the XML document. In this case the root is the application element. The target namespace is defined with the field xmlns:tns or targetNamespace whose types is xsd:anyURI.

The IDL module element has an identifier. Identifiers in RPCDDS IDL are defined:

< ID : ["a"-"z","A"-"Z", "\_"] (["a"-"z","A"-"Z","0"-"9","\_"])\* >

As the type of target namespace supports more characters there will be some rules to get the identifier of the IDL module element.

1. Everything before the final ‘/’ is not mapped.
2. All incompatible characters will be substituted by the character ‘\_’.

The following WADL example:

<?xml version="1.0"?>  
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"   
 xsi:schemaLocation=[http://wadl.dev.java.net/2009/02 wadl.xsd](http://wadl.dev.java.net/2009/02%20wadl.xsd)  
 xmlns:tns="urn:yahoo:yn"  
 xmlns:xsd=<http://www.w3.org/2001/XMLSchema>  
 xmlns="http://wadl.dev.java.net/2009/02">  
</application>

The RPCDDS IDL generated will be:

module urn\_yahoo\_yn  
{  
};

#### Documentation

#TODO

#### Grammars

The grammars element acts as a container for definitions of the format of data exchanged during execution of the protocol described by the WADL document. Such definitions may be included inline or by reference using the include element (see section *Include*).

No particular data format definition language is mandated. But RPCDDS imposes a limitation: W3C XML Schema must be used to define data format. (TODO: put reference) The appendix A describes how to map XML Schema to RPCDDS IDL.

##### Include

The include element allows the definitions of one or more data format descriptions to be included by reference. The href attribute provides a URI for the referenced definitions and is of type xsd:anyURI.

The include element will be mapped to the preprocessor tag #include. The URI of href attribute must reference a XML Schema file. As also assumes that this file has been mapped to RPCDDS IDL the preprocessor tag #include must reference the IDL file version. The following WADL example:

<?xml version="1.0"?>  
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"   
 xsi:schemaLocation=[http://wadl.dev.java.net/2009/02 wadl.xsd](http://wadl.dev.java.net/2009/02%20wadl.xsd)  
 xmlns:tns="urn:yahoo:yn"  
 xmlns:xsd=<http://www.w3.org/2001/XMLSchema>  
 xmlns="http://wadl.dev.java.net/2009/02">  
 <grammars>  
 <include  
 href="NewsSearchResponse.xsd"/>  
 <include  
 href="Error.xsd"/>  
 </grammars>   
</application>

The RPCDDS IDL generated will be:

#include “NewsSearchResponse.idl”

#include “Error.idl”

module urn\_yahoo\_yn  
{  
};

#### Resources

The resources element acts as a container for the resources provided by the application. A resources element has a base attribute of type xsd:anyURI that provides the base URI for each child resource identifier.

This WADL element only set the base URI. This element will be mapped as a #pragma directive. The name of this #pragma directive is RESOURCES\_BASE\_URI. This WADL element encapsulates all resources that the server will offer and these resources are linked with their methods. Then this WADL element is a good point to insert the IDL interface that the server offers.

The following WADL example:

<?xml version="1.0"?>  
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"   
 xsi:schemaLocation=[http://wadl.dev.java.net/2009/02 wadl.xsd](http://wadl.dev.java.net/2009/02%20wadl.xsd)  
 xmlns:tns="urn:yahoo:yn"  
 xmlns:xsd=<http://www.w3.org/2001/XMLSchema>  
 xmlns="http://wadl.dev.java.net/2009/02">  
 <grammars>  
 <include  
 href="NewsSearchResponse.xsd"/>  
 <include  
 href="Error.xsd"/>  
 </grammars>

<resources base="http://api.search.yahoo.com/NewsSearchService/V1/">

</resources>  
</application>

The RPCDDS IDL generated will be:

#include “NewsSearchResponse.idl”

#include “Error.idl”

module urn\_yahoo\_yn  
{

#pragma RESOURCES\_BASE\_URI <http://api.search.yahoo.com/NewsSearchService/V1/>

interface NewsSearchService\_V1

{

};  
};

#### Resource

A resource element describes a set of resources, each identified by a URI that follows a common pattern. A resource element has the following attributes:

* id – An optional attribute of type xsd:ID that identifies the resource element.
* path – An optional attribute of type xsd:string. If present, it provides a relative URI template for the identifier of the resource. The resource’s base URI is given by the resource elements’s parent resource of resources element.
* type – An optional attribute whose type is a space separated list of xsd:anyURI. Each value in the list is a cross reference (see section *Cross Referencing*) that identifies a resource-type element (see section ) that defines a set of methods.
* queryType – Defines the media type for the query component of the resource URI. Defaults to 'application/x-www-form-urlencoded' if not specified which results in query strings being formatted as specified in section 17.13 of HTML 4.01[[3](http://www.w3.org/Submission/wadl/#Xhtml401)]

### How WADL maps remote procedures into HTTP messages

## Protocol classes

Until now the encapsulation of the remote procedure and its parameters was responsible of the utility functions generated by the IDL compiler. These functions implemented for each remote procedure the following functionality:

* Encapsulate the remote procedure and its input parameters in a request. The type of this request is a DDS type. This functionality is used in the client side. In the example this type *depositRequest*:  
    
   struct depositRequest{

Identification clientServiceId; //@key

unsigned long numSec; //@key

Account ac;

long money;

}; //@top-level true

* Extract the remote procedure and its input parameters from a request. This functionality is used in the server side. In the example this type is *depositRequest*.
* Encapsulate the remote procedure and its output parameters in a reply. The type of this reply is a DDS type. This functionality is used in the server side. In the example this type *depositReply*:  
    
   struct depositReply{

Identification serverServiceId; //@key

Identification clientServiceId; //@key

unsigned long numSec; //@key

long ddsrpcRetCode;

ReturnCode deposit\_ret;  
 }; //@top-level true

* Extract the remote procedure and its output parameters from a reply. This functionality is used in the client side. . In the example this type *depositReply*.

Now there will be more than one serialization/deserialization mechanism. These mechanisms are called “protocols” in this document and each one is implemented in a class. These classes will offer the same interface and they are generated by the IDL compiler. The generated classes using the example will be:



The interface *BankProtocol* will be exposed by both protocols. The interface offers the functionality of the current utility functions. In the example the functionality is:

* Encapsulate *deposit* procedure and its input parameters in a request.
* Extract *deposit* procedure and its input parameters from a request.
* Encapsulate *deposit* procedure and its output parameters in a reply.
* Extract *deposit* procedure and its output parameters from a reply.

*BankDDSProtocol* will use the generated code by the application *rtiddsgen* to serialize/deserialize as it does right now. *BankRestfulProtocol* will use the API described in section (TODO: set section number) to serialize/deserialize.

### Hiding the type of the serialized data

All protocol classes use the class *Buffer* to hide the type of the buffer where data is serialized. The type of the data (DDS type or Resful serialized data) should be transparent outside the protocol classes and the transport classes.

A possible implementation would be that the *Buffer* class is a base class. This base class is used outside the protocol classes and transport classes. But this base class is inherited by classes that depend on the types they manage. As RPCDDS is implemented in C++, templates will be use to implement these classes. Instead of there are several classes, it will be one template class.

An example of how implement that will be shown. In the example *Buffer* class is the base class, and *Buffer\_* is the class template.

// An example type

typedef struct Prueba

{

int i;

} Prueba;

// Other example type

typedef struct Prueba2

{

char \*msg;

} Prueba2;

// Funtion that creates an instance of type 1.

Prueba\* createPrueba()

{

Prueba \*p = (Prueba\*)malloc(sizeof(Prueba));

return p;

}

// Funtion that deletes an instance of type 1.

void deletePrueba(Prueba \*p)

{

free(p);

}

// Funtion that creates an instance of type 2.

Prueba2\* createPrueba2()

{

Prueba2 \*p = (Prueba2\*)malloc(sizeof(Prueba2));

p->msg = (char\*)malloc(100);

return p;

}

// Funtion that deletes an instance of type 2.

void deletePrueba2(Prueba2 \*p)

{

free(p->msg);

free(p);

}

// Base class that will use outside protocol classes and transport classes.

class Buffer

{

public:

Buffer(){}

virtual ~Buffer(){}

};

// Template class that depends on the type.

template<typename T>

class Buffer\_ : public Buffer

{

public:

Buffer\_(T \*data, void (\*deleteFnc)(T \*data)) : m\_data(data),

m\_deleteFnc(deleteFnc)

{

}

// If object is destroyed, then the data too.

virtual ~Buffer\_()

{

m\_deleteFnc(m\_data);

}

private:

T \*m\_data;

void (\*m\_deleteFnc)(T \*data);

};

// Example that create a Buffer\_<Prueba> but return a Buffer object.

Buffer\* createBufferPrueba()

{

Prueba \*p = createPrueba();

Buffer\_<Prueba> \*b = new Buffer\_<Prueba>(p, deletePrueba);

return b;

}

// Example that create a Buffer\_<Prueba2> but return a Buffer object.

Buffer\* createBufferPrueba2()

{

Prueba2 \*p = createPrueba2();

Buffer\_<Prueba2> \*b = new Buffer\_<Prueba2>(p, deletePrueba2);

return b;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

Buffer \*b = createBufferPrueba();

Buffer \*b2 = createBufferPrueba2();

// Using dynamic\_cast, we can deduce the original type.

Buffer\_<Prueba> \*d = dynamic\_cast<Buffer\_<Prueba>\*>(b);

d = dynamic\_cast<Buffer\_<Prueba>\*>(b2);

return 0;

}

Using this example of the implementation, in the example of remote procedure the class *BankDDSProtocol* will use internally the classes *Buffer\_<depositeRequest>* and *Buffer\_<depositeReply>*.

In case of Restful, a buffer of bytes is always used for the serialization and deserialization. Then any Restful protocol, like *BankRestfulProtocol* class in the example, will use internally the same class *Buffer\_<char\*>*.

### Detecting the remote procedure

When the server is listening and it receives a request from a transport, the server doesn’t know which remote procedure the request is referring to. Protocol classes have to implement a function that detects the remote procedure from a serialized data. This function is included in the interface and it is *detect* function.

## Transport classes

The main functionalities of the transports are:

* Send data to other endpoint. This occurs when a client wants to send a request to the server or when a server wants to send a reply to the client.
* Receive data from other endpoint. This case occurs when a client wants to receive the reply from the server. In some transports could be needed the correlative request. This is the case of the transport that encapsulates DDS middleware. The other transports (like TCP transport for Restful) don’t need this information, but this information could be important for the future transports.
* Listen data from any endpoint. This case occurs when server is listening incoming requests. This call is blocking until a request will come.

The interface of all transports in next version of RPCDDS will be:



The functions of this interface are:

* *Listen*: Block the thread until a request arrives. This function not only returns the buffer that was received. This function also returns an object of class *Context*. This context is used by the server to determinate what transport it has to use to send the reply.
* *Send*: Sends the buffer to other endpoint. This function receives a second parameter. It can receive a context that has extra information about the destination. In the case the context indicates to use the Restful TCP transport, it contains the IP address of the destination.
* *Receive*: Receives a buffer with the reply. This reply has to be correlative to the input parameter of this function. This parameter will be the correlative request.

The class diagram of DDS transport and a TCP transport for Restful will be:



All classes related with the DDS transport encapsulate the current usage of DDS in the project RPCDDS. These classes will not be discussed in this document.

### Restful TCP transport

Main transport for Restful should be a TCP transport. This transport doesn’t need any generated code.

## Server’s proxy

The server’s proxy class is generated by the IDL compiler.



This class needs a protocol to know how serialize the remote procedure and it needs a transport to know how send the serialized data. An example of how a user will use this server’s proxy is:

Protocol \*protocol = new BankRestfulProtocol();

Transport \*transport = new TCPTransport("80.231.32.34:8080");

BankProxy \*proxy = new BankProxy(protocol, transport);

Account \*ac = AccountPluginSupport\_create\_data();

DDS\_Long money = 0;

ReturnCode deposit\_ret = SYSTEM\_ERROR;

try

{

deposit\_ret = proxy->deposit(ac, money);

}

catch(TimeoutException ex)

{

}

AccountPluginSupport\_destroy\_data(ac);

delete(proxy);

delete(transport);

delete(protocol);

An example of how the remote procedure in the example is implemented and how it use the protocol class and the transport class.

ReturnCode BankProxy::deposit(/\*in\*/ const Account\* ac, /\*in\*/ const DDS\_Long money)

{

ReturnCode returnedValue;

Buffer sBuffer = protocol->deposit\_serialize\_input(ac, money);

transport->send(sBuffer, NULL);

Buffer rBuffer = transport->receive(sBuffer);

protocol->deposit\_deserialize\_output(returnedValue);

return returnedValue;

}

## Server



Server can use several strategies to manage the incoming requests. User can decides what strategies to use in the server constructor. RPCDDS supports right now three strategies:

* Single thread strategy: The thread that got the request will process it.
* Thread pool strategy: A threadpool is used to process the incoming requests.
* Thread per request: For each incoming request a thread will be created and it will process the request.

Server supports several transports. Each transport will have an associated protocol. When a transport receives a request, the associated protocol will deserialize it. Then the reply will serialize by the associated protocol and it will send by the transport. *Context* object will be responsible for the reply is sent by the same transport.

For each transport a thread will be created. This thread will call to the *listen* function of the transport and it will be blocked.

# Behavior model

# Unit test design