eProsima RPC over DDS

User Manual
Version 1.0.3

The Middleware Experts

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 16 – Bajo K
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 DDS is licensed under the terms described in the RPCDDS_LICENSE file included in this distribution.

Technical Support

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

Table of Contents

1 Introduction

eProsima RPC over DDS is a high performance remote procedure call (RPC) framework. It combines a software stack with a code generation engine to build efficient services for several platforms and programming languages.

eProsima RPC over DDS uses the Data Distribution Service (DDS) standard from the Object Management Group (OMG) as the communication engine.

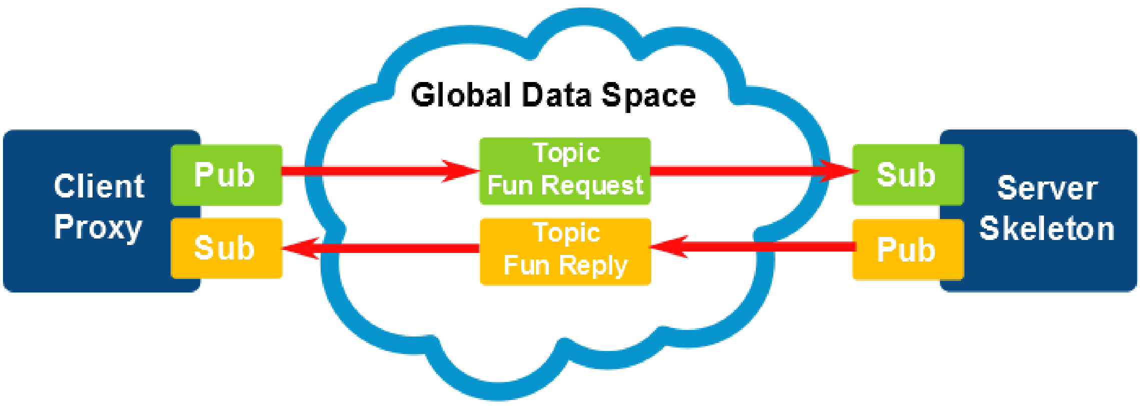
1.1 Client/Server communications over DDS

There are three main communication patterns used in distributed systems:

- Publish-Subscribe
- Request-Reply
- Point to Point

One example of Request-Reply pattern is the Remote Procedure Call (RPC). RPC allows an application to call a subroutine or procedure in another address space (commonly in another computer on a shared network).

The framework generates the Request-Reply code from the procedure definition using an Interface Definition Language (IDL), allowing the developer to focus in the application logic without bothering about the networking details.



1.2 *A quick example*

You write a .IDL file like this:

```
interface Example
{
    void exampleMethod();
};
```

Then you process the file with the *rpcddsgen* compiler to generate C++ code. Afterwards, you use that code to invoke remote procedures with the client proxy:

```
UDPProxyTransport *transport = new UDPProxyTransport(”ExampleService”);
ExampleProtocol *protocol = new ExampleProtocol();
ExampleProxy *proxy = new ExampleProxy(*transport, *protocol);
...
proxy->exampleMethod();
```

or to implement a server using the generated skeleton:

```
UDPServerTransport *transport = new UDPServerTransport(”ExampleService”);
ExampleProtocol *protocol = new ExampleProtocol();
SingleThreadStrategy *single = new SingleThreadStrategy();
ExampleServerImpl servant;
ExampleServer *server =
       new ExampleServer(*single, *transport, *protocol, servant);
...
server->serve();
```

See section 4.1 (Writing the IDL file) for a complete step by step example.

1.3 Main Features

- **Synchronous, asynchronous and one-way invocations**.
  - The synchronous invocation is the most common one. It blocks the client's thread until the reply is received from the server.
  - In the asynchronous invocation the request does not block the client's thread. Instead, the developer provides a callback object that is invoked when the reply is received.
  - The one-way invocation is a fire-and-forget invocation where the client does not care about the result of the procedure. It does not wait for any reply 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.
- **Several communications transports**:
  - Reliable and high performance UDP transport
  - NAT and firewall friendly TCP transport
  - Shared Memory transport.
- **Automatic Discovery**: The framework uses the underlying DDS discovery protocol to discover the different clients, servers and services.
- **Complete Publish/Subscribe Frameworks:** Users can integrate RPC over DDS Publish/Subscribe code in their applications.
- **High performance**: The framework uses a fast serialization mechanism that increases the performance.

2 Building an application

eProsima RPC over DDS 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. The 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. The developer just writes the behavior of the remote procedures using the generated skeleton.

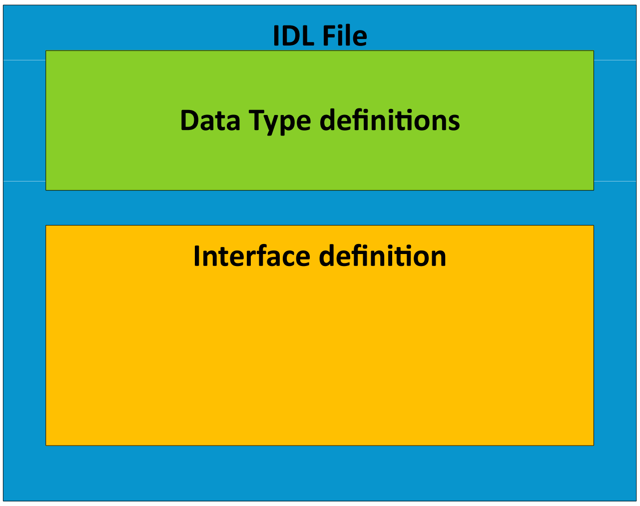
Steps to build an application:

- Define a set of remote procedures, using an Interface Definition Language.
- Using the provided IDL compiler, generate the specific remote procedure call support code (a Client Proxy and a Server Skeleton)
- Implement the server, filling the server skeleton with the behavior of the procedures.
- Implement the client, using the client proxy to invoke the remote procedures.

This section will describe the basic concepts of these four steps that a 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*

An Interface Definition Language (IDL) is used to define the remote procedures the server will offer. Data Types used as parameter types in these remote procedures are also defined in the IDL file. The IDL structure is based in OMG IDL and it is described in the following schema:



eProsima RPC over DDS includes a Java application named `rpcddsgen`. This application parses the IDL file and generates C++ code for the defined set of remote procedures. `rpcddsgen` application will be described in the section 2.2 (*Generating specific remote procedure call support code)*.

2.1.1 IDL Syntax and mapping to C++

2.1.1.1 Simple types

eProsima RPC over DDS supports a variety of simple types that the developer can use as parameters, returned values and members of complex types. The following tables show the supported simple types, how they are defined in the IDL file and what the `rpcddsgen` generates in C++ language.

TABLE 1: SPECIFYING SIMPLE TYPES IN IDL FOR C++ USING C++11 NATIVE TYPES

IDL Type	Sample in IDL File	Sample Output Generated by rpcddsgen
char	char char_member	char char_member
wchar	wchar wchar_member	wchar_t wchar_member
octet	octet octet_member	uint8_t octet_member
short	short short_member	int16_t short_member
unsigned short	unsigned short ushort_member	uint16_t ushort_member
long	long long_member	int32_t long_member
unsigned long	unsigned long ulong_member	uint32_t ulong_member
long long	long long llong_member	int64_t llong_member
unsigned long long	unsigned long long ullong_member	uint64_t ullong_member
float	float float_member	float float_member
double	double double_member	double double_member
boolean	boolean boolean_member	bool boolean_member
bounded string	string<20> string_member	std::string string_member /* maximum length = (20) */
unbounded string	string string_member	std::string string_member /* maximum length = (255) */

2.1.1.2 Complex types

Complex types can be created combining simple types. These complex types can be used as parameters or returned values. The following table shows the supported complex types, how they are defined in the IDL file and what `rpcddsgen` generates in C++ language.

TABLE 2: SPECIFYING COMPLEX TYPES IN IDL FOR C++ USING C++11 NATIVE TYPES

IDL Type	Sample in IDL File	Sample Output Generated by rpcddsgen

enum	enum PrimitiveEnum {
	    ENUM1,
	    ENUM2,
	    ENUM3
	};
	enum PrimitiveEnum {
	    ENUM1 = 10,
	    ENUM2 = 20,
	    ENUM3 = 30
	};
		enum PrimitiveEnum : uint32_t {
		    ENUM1,
		    ENUM2,
		    ENUM3
		};
		enum PrimitiveEnum : uint32_t {
		    ENUM1 = 10,
		    ENUM2 = 20,
		    ENUM3 = 30
		};

<table>
<tr><td>struct</td><td><pre>struct PrimitiveStruct {
    char char_member;
};</pre></td><td><pre>class PrimitiveStruct {
public:
    /** Constructors **/
    PrimitiveStruct();
    ...

    /** Getter and Setters **/
    char char_member();
    void char_member(char x);
    ...

private:
    char m_char_member;
};</pre></td></tr>
<tr><td>union</td><td><pre>union PrimitiveUnion
switch(long) {
    case 1:
        short short_member;
    default:
        long long_member;
};</pre></td><td><pre>class PrimitiveUnion {
public:
    /** Constructors **/
    PrimitiveStruct();
    ...

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

    /** Getter and Setters **/
    int16_t short_member();
    int32_t long_member();
    ...

private:
    int32_t m__d;
    int16_t m_short_member;
    int32_t m_long_member;
};</pre></td></tr>
<tr><td>typedef</td><td><pre>typedef short TypedefShort;</pre></td><td><pre>typedef int16_t TypedefShort;</pre></td></tr>
<tr><td>array
(See note below)</td><td><pre>struct OneDArrayStruct {
    short short_array[2];
};
struct TwoDArrayStruct {
    short short_array[1][2];
};</pre></td><td><pre>class OneDArrayStruct {
    ...

private:
    std::array<int16_t, 2>
        m_short_array;
};

class TwoDArrayStruct {
    ...

private:

std::array<std::array<int16_t,
2>, 1> m_short_array;
};</pre></td></tr>
<tr><td>bounded
sequence
(See note below)</td><td><pre>struct SequenceStruct {
    sequence<short,4>
        short_sequence;
};</pre></td><td><pre>class SequenceStruct {
    ...

private:
    std::vector<int16_t>
        m_short_sequence;
};</pre></td></tr>
</table>

unbounded sequence (See note below)	`struct SequenceStruct {`
	`    sequence<short>`
	`        short_sequence;`
	`};`
		`class SequenceStruct {`
		`    ...`
		
		`private:`
		`    std::vector<int16_t>`
		`        m_short_sequence;`
		`};`

Note: These complex types cannot be used directly as procedure's parameter. In these cases, a typedef has to be used to redefine them.

2.1.1.3 Parameter definition

There are three reserved words that are used in the procedure's parameter definitions. It is mandatory to use one of them in each procedure's parameter definition. The following table shows these reserved words and their meaning:

Reserved word	Meaning
in	The parameter is an input parameter.
inout	The parameter acts as an input and output parameter.
output	The parameter is an output parameter.

Suppose the type `T` is defined as the type of the parameter. If the parameter uses the reserved word `in` and the type `T` is a simple type or an enumeration, then the type is mapped in C++ as `T`. In the case the type `T` is a complex type, the type is mapped in C++ as `const T&`. If the parameter uses the reserved word `inout` or `out`, then the type is mapped in C++ as `T&`.

As it was commented in section 2.1.1.2 *(Complex types)*, array and sequence types cannot be directly defined as parameter types. To do so, they have to be previously redefined using a `typedef`. This redefinition can be used as a parameter.

2.1.1.4 Function definition

A procedure's definition is composed of two or more elements:

- The type of the returned value. `void` type is allowed.
- The name of the procedure.
- A list of parameters. This list could be empty.

An example of how a procedure should be defined is shown below:

```
long funcName(in short param1, inout long param2);
```

`rpcddsgen` application maps the functions following these rules:

- The type of the C++ returned value is the same as the one defined in the IDL file, using the tables described in sections 2.1.1.1 (Simple types) and 2.1.1.2 (Complex types) for the mapping.
- The name of the C++ function is the same as the name of the defined function in the IDL file.
- The order of the parameters in the C++ function is the same as the order in the IDL file. The parameters are mapped in C++ as it was described in section 2.1.1.3 (*Parameter definition)*.

Following these rules, the previous example would generate one of the following C++ functions, depending on the chosen types:

```
int32_t funcName(int16_t param1, int32_t& param2);
```

2.1.1.5 Exception definition

IDL functions can raise user-defined exceptions to indicate the occurrence of an error. An exception is a structure that may contain several fields. An example of how to define an exception is shown below:

```
exception ExceptionExample
{
    long count;
    string msg;
};
```

This example would generate one of the following C++ exceptions, depending on the chosen types:

```
class ExceptionExample: public eprosima::rpc::exception::UserException
{
public:
    ExceptionExample();
    ExceptionExample(const ExceptionExample &ex);
    ExceptionExample(ExceptionExample&& ex);
    ExceptionExample& operator=(const ExceptionExample &ex);
    ExceptionExample& operator=(ExceptionExample&& ex);
    virtual ~ExceptionExample() throw();
    virtual void raise() const;

    /** Getters and Setters **/
    int32_t count() const;
    int32_t& count();
    void count(int32_t _count);
    ...

private:
    /** Exception members **/
    int32_t m_count;
    std::string m_msg;
};
```

To specify that an operation can raise one or more user-defined exceptions, first define the exception and then add an IDL raises clause to the operation definition, like this example does:

```
exception Exception1
{
    long count;
};

exception Exception2
{
    string msg;
};

void exceptionFunction()
    raises(Exception1, Exception2);
```

2.1.1.6 Interface definition

The remote procedures that the server will offer have to be defined in an IDL interface. An example of how an interface should be defined is shown:

```
interface InterfaceExample
{
    // Set of remote procedures.
};
```

The IDL interface will be mapped in three classes:

- `InterfaceExampleProxy`: A local server's proxy that offers the remote procedures to the client application. Client application must create an object of this class and call the remote procedures.
- `InterfaceExampleServerImpl`: This class contains the remote procedures definitions. These definitions must be implemented by the developer. *eProsima RPC over DDS* creates one object of this class. It is used by the server.
- `InterfaceExampleServer`: The server implementation. This class executes a server instance.

eProsima RPC over DDS supports interface inheritance, like the following example shows:

```
interface ParentInterface
{
    void function1();
};

interface ChildInterface : ParentInterface
{
    void function2();
};
```

In this example, the IDL interface `ChildInterface` has two functions: `function1` and `function2`.

2.1.1.7 Module definition

To group related definitions, such as complex types, exceptions, functions and interfaces, a developer can use modules:

```
module ModuleExample
{
    // Set of definitions
};
```

A module will be mapped into a C++ namespace, and every definition inside it will be defined within the generated namespace in C++.

2.1.1.8 Limitations

`rpcddsgen` application has some limitations concerning IDL syntax:

- Two procedures cannot have the same name.
- Complex types (array and sequences) used in procedure definitions must be previously named using `typedef` keyword, as CORBA IDL 2.0 specification enforces.
- Using DDS types, a function cannot have an array as returned type.

2.1.2 Example

This example will be used as a base to other examples in the following sections. IDL syntax described in the previous subsection is shown through an example:

```
// 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);
};
```

2.2 *Generating specific remote procedure call support code*

Once the API is defined in a IDL file, we need to generate code for a client proxy and a server skeleton. *eProsima RPC over DDS* provides the `rpcddsgen` tool for this purpose: it parses the IDL file and generates the corresponding supporting code.

2.2.1 RPCDDSGEN Command Syntax:

The general syntax is:

```
rpcddsgen [options] <IDL file> <IDL file> ...
```

Options:

Option	Description
-help	Shows help information.
-version	Shows the current version of *eProsima RPC over DDS*
-ppPath <directory>	Location of the C/C++ preprocessor.
-ppDisable	Disables the C/C++ preprocessor. Useful when macros or includes are not used.
-replace	Replaces existing generated files.
-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
-d <path>	Sets an output directory for generated files
-t <temp dir>	Sets a specific directory as a temporary directory
-transport <transport>	Select the DDS transport that generated code will use. Possible values: rti, rtps. Default: rti
-topicGeneration	Defines how DDS topics are generated.

<option>	**Possible values:** byInterface, byOperation. **Default:** byInterface

The rpcddsgen application generates several files. They will be described in this section. Their names are generated using the IDL file name. The *<IDLName>* tag has to be substituted by the file name.

2.2.2 Server side

rpcddsgen generates C++ header and source files with the declarations and the definitions of the remote procedures. These files are the skeletons of the servants that implement the defined interfaces. The developer can use each definition in the source files to implement the behavior of the remote procedures. These files are *<IDLName>*ServerImpl.h and *<IDLName>*ServerImpl.cxx. rpcddsgen also generates a C++ source file with an example of a server application and a server instance. This file is *<IDLName>*ServerExample.cxx.

2.2.3 Client side

rpcddsgen 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 *<IDLName>*ClientExample.cxx.

2.3 Server implementation

After the execution of rpcddsgen, two files named *<IDLName>*ServerImpl.cxx and *<IDLName>*ServerImpl.h will be generated. These files are the skeleton of the interfaces 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 *deposit* seen in our Example, the possible generated definitions are:

```
ReturnCode BankServerImpl::deposit(/*in*/const Account& ac, /*in*/ int32_t
money)
{
   ReturnCode returnedValue = SYSTEM_ERROR;

   return returnedValue;
}
```

Keep in mind a few things when this servant is implemented.

- in parameters can be used by the developer, but their allocated memory cannot be freed, either any of their members.
- inout parameters can be modified by the developer, but before allocate memory in their members, old allocated memory has to be freed.
- out parameters are not initialized. The developer has to initialize them.

The code generated by rpcddsgen also contains the server classes. These classes are implemented in the files *<IDLName>*Server.h and *<IDLName>*Server.cxx. They offer the resources implemented by the servants.

When an object of the class *<IDLName>*Server is created, proxies can establish a connection with it. How this connection is created and how the proxies find the server depends on the selected network transport. These transports are described in section 3.1 (*Network transports)*.

2.3.1 API

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

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

    virtual ~BankServer();
    ...
};
```

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 the section 3.4.1 (Single thread strategy).

The second parameter expects the network transport to connect with client proxies. The third parameter is the protocol. It's generated by *rpcddsgen* 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.

2.3.2 Exceptions

In the server side, developers can inform about an error in the execution of the remote procedures. The exception `eprosima::rpc::exception::ServerInternalException` can be thrown in the developer's code. This exception will be delivered to the proxy and will be thrown in the client side. Examples of how this exception can be thrown are shown below:

```
ReturnCode BankServerImpl::deposit(/*in*/const Account& ac, /*in*/ int32_t
money)
{
   ReturnCode returnedValue = SYSTEM_ERROR;

   throw eprosima::rpc::exception::ServerInternalException("Error in deposit
procedure");

   return returnedValue;
}
```

2.3.3 Example

Using the suggested IDL Example, the developer can create a server in the following way:

```
unsigned int threadPoolSize = 5;
ThreadPoolStrategy *pool = NULL;
BankProtocol *protocol = NULL;
UDPServerTransport *transport = NULL;
BankServer *server = NULL;
BankServerImplExample servant;

try
{
   pool = new ThreadPoolStrategy(threadPoolSize);
   transport = new UDPServerTransport("MyBankName");
   protocol = new BankProtocol();
   server = new BankServer(*pool, *transport, *protocol, servant);
   server->serve();
}
catch(eprosima::rpc::exception::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

2.4 Client implementation

The code generated by `rpcddsgen` contains classes that act like proxies of the remote servers. These classes are implemented in the files `<IDLName>Proxy.h` and `<IDLName>Proxy.cxx`. The proxies offer the resources from the servers, so the developer can directly invoke its remote procedures.

2.4.1 API

Using the suggested IDL Example, the API of this class is:

```
class BankProxy : public eprosima::rpc::proxy::Proxy
{
    public:

        BankProxy(eprosima::rpc::transport::ProxyTransport &transport,
            eprosima::rpc::protocol::BankProtocol &protocol);

        virtual ~BankProxy();

        ReturnCode deposit(/*in*/ const Account& ac, /*in*/ int32_t money);

        void deposit_async(Bank_depositCallbackHandler &obj, /*in*/ const
Account& ac, /*in*/ int32_t money);

};
```

The proxy provides a constructor. It expects the network transport as the first parameter. The second parameter is the protocol. Again, it is generated by `rpcddsgen` and its duty is to serialize and deserialize protocol data.

The proxy provides the remote procedures to the developer. Using the suggested IDL, our proxy will provide the remote procedure `deposit`. The function `deposit_async` is the asynchronous version of the remote procedure. Asynchronous calls are described in the section 3.2 (Asynchronous calls).

2.4.2 Exceptions

While a remote procedure call is executed, an error can occur. In these cases, exceptions are used to report errors. Following exceptions can be thrown when a remote procedure is called:

Exception	Description
eprosima::rpc::exception::ClientInternalException	This exception is thrown when there is a problem in the client side.
eprosima::rpc::exception::ServerTimeoutException	This exception is thrown when the maximum time was exceeded waiting the server's reply.
eprosima::rpc::exception::ServerInternalException	This exception is thrown when there is a problem in the server side.
eprosima::rpc::exception::ServerNotFoundException	This exception is thrown when the proxy cannot find any server.

All exceptions have the same base class: `eprosima::rpc::exception::Exception`.

2.4.3 Example

Using the suggested IDL example, the developer can access to the `deposit` remote procedure the following way:

```
BankProtocol *protocol = NULL;
UDPProxyTransport *transport = NULL;
BankProxy *proxy = NULL;

try {
   protocol = new BankProtocol();
   transport = new UDPProxyTransport("MyBankName");
   proxy = new BankProxy(*transport, *protocol);
}
catch(eprosima::rpc::exception::InitializeException &ex) {
   std::cout << ex.what() << std::endl;
}

Account ac;
int32_t money;
ReturnCode depositRetValue;

try {
   depositRetValue = proxy->deposit(ac, money);
}
catch(eprosima::rpc::exception::Exception &ex) {
   std::cout << ex.what() << std::endl;
}
```

3 Advanced concepts

3.1 Network transports

eProsima RPC over DDS provides a network transport implemented using eProsima FastRTPS library.

3.1.1 RTPS Transport

The purpose of this transport is to create a connection between a proxy and a server located in the same local network. Instead of being implemented using RTI DDS middleware, this transport is implemented only using the RTPS-level. It is implemented by two classes. One is used by proxies and the other is used by servers.

To use this network transport, you have to use the option `-transport rtps` in *fastrpcgen* Java application.

RTPSProxyTransport

`RTPSProxyTransport` class implements an RTPS transport to be used by proxies:

```
class RTPSProxyTransport: public ProxyTransport
{
   public:
      RTPSProxyTransport(std::string remoteServiceName, std::string
instanceName, int domainId = 0, long timeout = 10000L);
      virtual ~RTPSProxyTransport();
};
```

This class has one constructor. It sets the transport to use RTPS-level discovery mechanism. This discovery mechanism allows the proxy to find any server in the local network. There are three potential scenarios:

- In the local network there is not any server using the provided service name. In this case, the proxy will not create any connection until a server announces to the network. If a client tries to invoke a remote procedure before this happens, it will raise a `ServerNotFoundException`.
- In the local network there is only one server using the provided service name. When a proxy is created, it will find the server and will create a connection channel with it. When the client application uses the proxy to call a remote procedure, this server will execute this procedure and return the reply from the server.
- In the local network there are several servers using the same service name. This scenario could occur when the user wants to have redundant servers to avoid failures in the system. When a proxy is created, it will find all servers and will create a connection channel with each one. When the client application uses the proxy to call a remotely procedure, all servers will execute the procedure but the client will receive only one reply from one server.

Using the suggested IDL example, the developer could create a proxy that connects with a specific server in a local network:

```
BankProtocol *protocol = NULL;
RTPSProxyTransport *transport = NULL;
BankProxy *proxy = NULL;

try
{
   protocol = new BankProtocol();
   transport = new RTPSProxyTransport("MyBankName");
   proxy = new BankProxy(*transport, *protocol);
}
catch(eprosima::rpc::exception::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}

Account ac;
int32_t money;
ReturnCode depositRetValue;

try
{
   depositRetValue = proxy->deposit(ac, money);
}
catch(eprosima::rpc::exception::Exception &ex)
{
   std::cout << ex.what() << std::endl;
}
```

RTPSServerTransport

`RTPSServerTransport` class implements an RTPS transport to be used by servers.

```
class RTPSServerTransport: public ServerTransport
{
   public:
      RTPSServerTransport(std:: string serviceName, std::string instanceName,
int domainId = 0);
      virtual ~RTPSServerTransport();
};
```

This class has one constructor. This constructor sets the transport to use RTPS discovery mechanism. RTPS discovery mechanism allows the server to discover any proxy in the local network.
Using the suggested IDL example, the developer could create a server with this code:

```
unsigned int threadPoolSize = 5;
ThreadPoolStrategy *pool = NULL;
BankProtocol *protocol = NULL;
RTPSServerTransport *transport = NULL;
BankServer *server = NULL;
BankServerImplExample servant;
```

```
try
{
   pool = new ThreadPoolStrategy(threadPoolSize);
   transport = new RTPSServerTransport("MyBankName", "");
   protocol = new BankProtocol();
   server = new BankServer(*pool, *transport, *protocol, servant);
   server->serve();
}
catch(eprosima::rpc::exception::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

3.2 *Asynchronous calls*

eProsima RPC over DDS supports asynchronous calls: a client application can call a remote procedure and that call does not block the thread execution.

3.2.1 Calling a Remote procedure asynchronously

`rpcddsgen` generates one asynchronous call for each remote procedure. These methods are named `<RemoteProcedureName>_async`. They receive as parameters the object that will be called when request arrives and the input parameters of the remote procedure. Using the IDL example, `rpcddsgen` will generate next asynchronous method in the proxy:

```
void deposit_async(Bank_depositCallbackHandler &obj, /*in*/ const Account& ac,
/*in*/ int32_t money);
```

The asynchronous version of the remote procedures can also generate exceptions. The exceptions that could be thrown are:

Exception	Description
eprosima::rpc::exception::ClientException	This exception is thrown when there is a problem in the client side.
eprosima::rpc::exception::ServerNotFoundException	This exception is thrown when the proxy cannot find any server.

Example:

```
class Bank_depositHandler: public depositCallbackHandler
{
    void deposit(/*out*/ ReturnCode deposit_ret)
    {
        // Client desired behaviour when the reply arrives
    }

    virtual void on_exception(const eprosima::rpc::exception::Exception &ex)
    {
        // Client desired behaviour on exception
    }
}


void main()
```

```
{
   UDPProxyTransport *transport = NULL;
   BankProtocol *protocol = NULL;
   BankProxy *proxy = NULL;

   try
   {
      transport = new UDPProxyTransport("MyBankName");
      protocol = new BankProtocol();
      proxy = new BankProxy(*transport, *protocol);
   }
   catch(eprosima::rpc::exception::InitializeException &ex)
   {
      std::cout << ex.what() << std::endl;
   }

   Account ac;
   int32_t money = 0;
   Bank_depositHandler deposit_handler;

   try
   {
      proxy->deposit_async(deposit_handler, ac, money);
   }
   catch(eprosima::rpc::exception::Exception &ex)
   {
      std::cout << ex.what() << std::endl;
   }
}
```

3.2.2 Reply Call-back object

The client is notified of the reply through an object that the developer passes as a parameter to the asynchronous call. `rpcddsgen` generates one abstract class for each remote procedure the user will use in asynchronous calls. These classes are named `<InterfaceName>_<RemoteProcedureName>CallbackHandler`. Two abstract methods are created inside these classes. One is called when the reply arrives. This function has as parameter the return value of the remote procedure. The other function is called in case of exception. The user should create a class that inherits from `<InterfaceName>_<RemoteProcedureName>CallbackHandler` class and then implement both methods. Using the IDL example, `rpcddsgen` will generate this class:

```
class Bank_depositCallbackHandler
{
public:
    virtual void deposit( /*out*/ ReturnCode deposit_ret) = 0;
    virtual void error(const eprosima::rpc::exception::Exception &ex) = 0;
};
```

The function that is called in case of exception could receive these exceptions:

Error code	Description
eprosima::rpc::exception::ClientInternalException	An exception occurs in the client side.
eprosima::rpc::exception::ServerTimeoutException	The maximum time was exceeded waiting the server's reply.
eprosima::rpc::exception::ServerInternalException	An exception occurs in the server side.

3.3 One-way calls

Sometimes a remote procedure doesn't need the reply from the server. For these cases, *eProsima RPC over DDS* supports one-way calls.

A developer can define a remote procedure as one-way, and when the client application calls the remote procedure, the thread does not wait for any reply.

To create a one-way call, the remote procedure has to be defined in the IDL file with the following rules:

- The `oneway` reserved word must be used before the method definition.
- The returned value of the method must be the `void` type.
- The method cannot have any `inout` or `out` parameter.

An example of how a one-way procedure has to be defined using IDL is shown below:

```
interface Bank
{
	oneway void deposit(in Account ac, in long money);
};
```

3.4 Threading Server strategies

eProsima RPC over DDS library provides several threading strategies for the server. This subsection describes these strategies.

3.4.1 Single thread strategy

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

```
SingleThreadStrategy *single = NULL;
BankProtocol *protocol = NULL;
UDPServerTransport *transport = NULL;
BankServer *server = NULL;
BankServerImplExample servant;

try
{
   single = new SingleThreadStrategy();
   transport = new UDPServerTransport("MyBankName");
   protocol = new BankProtocol();
   server = new BankServer(*single, *transport, *protocol, servant);
   server->serve();
}
catch(eprosima::rpc::exception::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

3.4.2 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 in the thread pool.

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

```
unsigned int threadPoolSize = 5;
ThreadPoolStrategy *pool = NULL;
BankProtocol *protocol = NULL;
UDPServerTransport *transport = NULL;
BankServer *server = NULL;
BankServerImplExample servant;

try
{
   pool = new ThreadPoolStrategy(threadPoolSize);
   transport = new UDPServerTransport("MyBankName");
   protocol = new BankProtocol();
   server = new BankServer(*pool, *transport, *protocol, servant);
   server->serve();
}
catch(eprosima::rpc::exception::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

3.4.3 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 a `ThreadPerRequestStrategy` object in the constructor method.

```
ThreadPerRequestStrategy *perRequest = NULL;
BankProtocol *protocol = NULL;
UDPServerTransport *transport = NULL;
BankServer *server = NULL;
BankServerImplExample servant;

try
{
   perRequest = new ThreadPerRequestStrategy();
   transport = new UDPServerTransport("MyBankName");
   protocol = new BankProtocol();
   server = new BankServer(*perRequest, *transport, *protocol, servant);
   server->serve();
}
catch(eprosima::rpc::exception::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

4 HelloWorld example

In this section an example is shown step by step. This example has one remote procedure. A client can invoke this procedure by passing a string with a name as parameter. The server returns a new string that appends the name to a greeting sentence.

4.1 *Writing the IDL file*

Write a simple interface named `HelloWorld` that has a `hello` method. Store this IDL definition in a file named `HelloWorld.idl`

```
// HelloWorld.idl

interface HelloWorld
{
	string hello(in string name);
};
```

4.2 *Generating specific code*

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

```
rpcddsgen -example x64Win64VS2010 HelloWorld.idl
```

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

```
rpcddsgen -example x64Linux2.6gcc4.4.5 HelloWorld.idl
```

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

This command 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 is 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;
    UDPProxyTransport *transport = NULL;
    HelloWorldProxy *proxy = NULL;

    // Creation of the proxy for interface "HelloWorld".
    try
    {
        protocol = new HelloWorldProtocol();
        transport = new UDPProxyTransport("HelloWorldService");
        proxy = new HelloWorldProxy(*transport, *protocol);
    }
    catch(InitializeException &ex)
    {
        std::cout << ex.what() << std::endl;
        return -1;
    }

    // Create and initialize parameters.
    std::string  name = "Richard";

    // Create and initialize return value.
    std::string  hello_ret = "";


    // Call to remote procedure "hello".
    try
    {
        hello_ret = proxy->hello(name);
    }
    catch(SystemException &ex)
    {
        std::cout << ex.what() << std::endl;
    }

    std::cout << hello_ret << std::endl;

    delete proxy;
    delete transport;
    delete protocol;

    return 0;
}
```

4.4 Server implementation

`rpcddsgen` 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 with a greeting sentence. Open the file and copy this code:

```
#include "HelloWorldServerImpl.h"

std::string HelloWorldServerImpl::hello(/*in*/ const std::string &name)
{
  std::string hello_ret;

  // Create the greeting sentence.
  hello_ret = “Hello “ + name + “!”;

  return hello_ret;
}
```

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:

```
Hello Richard!
```

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.