eProsima RPC over DDS

Users Manual
Version 0.1.RC1

Experts in networking middleware
eProsima © 2012

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

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 services that work efficiently in serveral platforms and programming languages.

eProsima RPC over DDS uses the Data Distribution Service (DDS) standard from the Object Management Group (OMG) as the communications engine to transmit the requests and the replays of remote procedure calls.

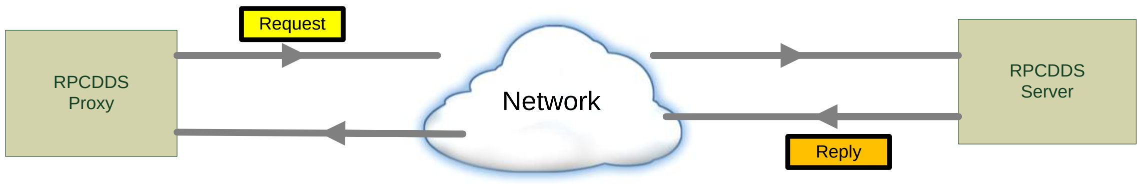
1.1 *Client/Server communications over DDS*

Distributed applications usually follow a communication pattern or paradigm to interact between them. Actually there are three main patterns used in distributed systems:

- Publish/Subscribe
- Client/Server
- Peer to Peer (P2P)

One example of client/server paradigm is the Remote Procedure Call (RPC). RPC allows an application to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network).

eProsima RPC over DDS provides an implementation of this general concept of invoking remote procedures. eProsima RPC over DDS is a service invocation framework that enables developers to build distributed applications with minimal effort. It makes transparent the remote procedure call to developer without the programmer explicitly coding the details for this remote interaction and allows developers to focus his efforts on their application logic.



1.2 Main Features

eProsima RPC over DDS provides an easy way to invoke remote proceduresand a high performance and reliable communications engine (DDS).

eProsima RPC over DDS also exposes these features:

- **Synchronous, asynchronous and one-way invocations**.
    - The synchronous invocation is the common invocation and it blocks the client's thread until the reply is received from the server.
    - The asynchronous invocation sends the request to the server but it doesn't blocks the client's thread. In the asynchronous invocation the developer provides a callback object that will be invoked when the reply is received from the server.
    - The one-way invocation is a fire-and-forget invocation where the client does not care about the success or failure of the invocation. The one-way invocation does not expect any reply from the server.
- **Different threading strategies for the server**. These strategies define how the server acts when a new request is received. Current supported strategies are: single-thread strategy, thread-pool strategy and thread-per-request strategy.
    - Single-thread strategy uses one thread for all incoming requests.
    - Thread-pool strategy uses thread-pool's threads to process the incoming requests.
    - Thread-per-request strategy creates a new thread for each new incoming request and this new thread will process the request.
- **Several communications transports**:
    - Reliable and high performance UDP transport
    - NAT and firewall friendly TCP transport
    - Shared Memory transport.
- **Automatic Discovery**: The framework makes use of the underlying DDS discovery protocol to discover the different clients, servers and services.
- **Complete Publish/Subscribe Frameworks:** Users can mix RPC with DDS Publish/Subscribe code in their applications.

2 Building an application

eProsima RPC over DDS allows a developer to implement easily a distributed application using remote procedure invocations. In this paradigm a server offers a set of remote procedures that the client can call remotely. How the clients call these remote procedures should be transparent for the developer. From the point of view of the developer, a proxy object that represents the remote server could be created in his application and this object would offer the set of remote procedures that the server implements. In the same way, how server obtains a request from the network and sends the reply should be transparent for the developer. Only the implementation of remote procedures concerns to the developer.

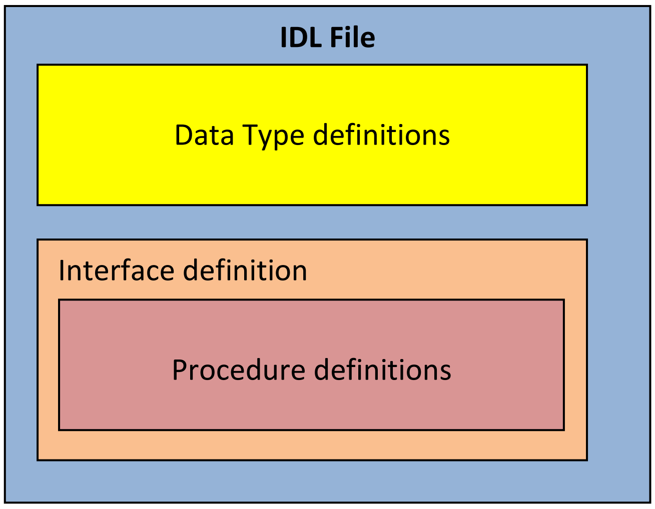
eProsima RPC over DDS offers this transparency to the developer and facilitates the development.
The general steps to build an application are:

- Define a set of remote procedures, using an Interface Definition Language.
- Generation of specific remote procedure call support code: a Client Proxy and a Server Skeleton.
- Implement the server: Fill the server skeleton with the procedures behavior.
- Implement the client: Use 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 its distributed application. Advanced concepts are described in section *Advanced concepts*.

2.1 *Defining a set of remote procedures*

Interface Definition Language (IDL) is used by eProsima RPC over DDS to define the remote procedures that server will offer to clients. Data Type definitions used for parameters in these remote procedures are also defined in the IDL file. The IDL structure is based in CORBA 2.x 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 specific set of remote procedures that the developer has defined. `rpcddsgen` application will be described in the section *Generating specific remote procedure call support code*.

eProsima RPC over DDS uses the same data types of its communications engine, DDS, enabling the use of both technologies in the same application.

2.1.1 IDL Syntax and mapping to C++

2.1.1.1 Simple types

eProsima RPC for DDS supports a variety of simple types that the developer can use in the procedure's parameters, returned values and in the definition of complex types. The following table shows 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++

IDL Type	Sample in IDL File	Sample Output Generated by rpcddsgen
char	`char char_member`	`DDS_Char char_member`
wchar	`wchar wchar_member`	`DDS_Wchar wchar_member`
octet	`octet octet_member`	`DDS_Octet octet_member`
short	`short short_member`	`DDS_Short short_member`
unsigned short	`unsigned short ushort_member`	`DDS_UnsignedShort ushort_member`
long	`long long_member`	`DDS_Long long_member`
unsigned long	`unsigned long ulong_member`	`DDS_UnsignedLong ulong_member`
long long	`long long llong_member`	`DDS_LongLong llong_member`
unsigned long long	`unsigned long long ullong_member`	`DDS_UnsignedLongLong ullong_member`
float	`float float_member`	`DDS_Float float_member`
double	`double double_member`	`DDS_Double double_member`
boolean	`boolean boolean_member`	`DDS_Boolean boolean_member`
bounded string	`string<20> string_member`	`char* string_member` `/* maximum length = (20) */`
unbounded string	`string string_member`	`char* string_member` `/* maximum length = (255) */`
bounded wstring	`wstring<20> wstring_member`	`DDS_Wchar* wstring_member` `/* maximum length = (20) */`
unbounded wstring	`wstring wstring_member`	`DDS_Wchar* wstring_member` `/* maximum length = (255) */`

2.1.1.2 Complex types

Complex types can be created by the developer using simple types. These complex types can be used as procedure's 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++

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`
	`};`
		`typedef enum PrimitiveEnum {`
		`    ENUM1,`
		`    ENUM2,`
		`    ENUM3`
		`} PrimitiveEnum;`
		`typedef enum PrimitiveEnum {`
		`    ENUM1 = 10,`
		`    ENUM2 = 20,`
		`    ENUM3 = 30`
		`} PrimitiveEnum;`

struct	`struct PrimitiveStruct {`
	`    char char_member;`
	`};`
		`typedef struct PrimitiveStruct {`
		`    DDS_Char char_member;`
		`} PrimitiveStruct;`

union	`union PrimitiveUnion switch(long)`
	`{`
	`    case 1:`
	`        short short_member;`
	`    default:`
	`        long longt_member;`
	`};`
		`typedef struct PrimitiveUnion {`
		`    DDS_Long _d;`
		`    struct {`
		`        short short_member;`
		`        long longt_member;`
		`    } _u;`
		`} PrimitiveUnion;`

typedef	`typedef short TypedefShort;`	`typedef DDS_Short TypedefShort;`

array
(See note below)
	`struct OneDArrayStruct {`
	`    short short_array[2];`
	`};`
	`struct TwoDArrayStruct {`
	`    short short_array[1][2];`
	`};`
		`typedef struct OneDArrayStruct {`
		`    DDS_Short short_array[2];`
		`} OneDArrayStruct;`
		`typedef struct TwoDArrayStruct {`
		`    DDS_Short short_array[1][2];`
		`} TwoDArrayStruct;`

bounded sequence
(See note below)
	`struct SequenceStruct {`
	`    sequence<short,4>`
	`        short_sequence;`
	`};`
		`typedef struct SequenceStruct {`
		`    DDSShortSeq short_sequence;`
		`} SequenceStruct;`

unbounded sequence
(See note below)
	`struct SequenceStruct {`
	`    sequence<short>`
	`        short_sequence;`
	`};`
		`typedef struct SequenceStruct {`
		`    DDSShortSeq short_sequence;`
		`} SequenceStruct;`

Note: These complex types cannot be used directly as procedure's parameter. In these cases, a typedef has to be use 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 three reserved words and their meaning:

Reserved word	Meaning
in	This reserved word specifies that the procedure's parameter is an input parameter.
Inout	This reserved word specifies that the procedure's parameter acts as input and output parameter.
output	This reserved word specifies that the procedure's parameter is only 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&`. For the type of the returned value of the procedure, it is mapped in C++ as `T`.
As was commented in section *Complex types*, array and sequence types cannot be defined as a parameter type directly. To redefine these types it must be used a `typedef` and use it as parameter type.

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:

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

`rpcddsgen` application maps the functions following these rules:

- The type of the returned value is mapped in C++ as it was described in section *Parameter definition*.
- Name of the C++ function is the same as the name of the defined function in IDL.
- The order of the parameters in the C++ function is the same as in defined function. The parameters are mapped in C++ as it was described in section *Parameter definition*.

Following these rules the previous example would generate next C++ function:

```
DDS_Long funcName(DDS_Short param1, DDS_Long& param2);
```

2.1.1.5 Interface definition

The set of remote procedures that the server will offer has to be encapsulated by 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 should create an object of this class and call the remote procedures.
- `InterfaceExampleServerImpl`: This class contains the remote procedures definitions. These definitions should be implemented by the developer. RPCDDS creates only one object of this class and this object is used by the server.
- `InterfaceExampleServer`: The server implementation. This class executes a server instance.

2.1.1.6 Limitations

`rpcddsgen` application has several limitations about IDL syntax:

- `rpcddsgen` can handle just one interface per IDL file.
- Type definitions must be declared before the interface.
- Two procedures cannot have the same name.
- **The interface and the IDL file must 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.
- No namespace (`module` keyword) support in this release.

2.1.2 Example

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);
};
```

This example will be used as base of other examples in next sections.

2.2 *Generating specific remote procedure call support code*

Once the procedures are defined in a IDL file, we need to generate code for a client proxy and a server. eProsima RPC over DDS provides the `rpcddsgen` tool to accomplish this task: 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>
```

Options:

Option	Description
-ppPath <directory>	Location of the C/C++ preprocessor.
-ppDisable	Indicates that C/C++ preprocessor has not to be used.
-replace	Replace generated files.
-example <platform>	Creates a solution in the specific platform. This solution will be use by the developer to compile the client and the server. **Possible values:** `i86Win32VS2010, x64Win64VS2010, i86Linux2.6gcc4.4.3, x64Linux2.6gcc4.5.1`
-version	Shows the version of eProsima RPC over DDS

NOTE: Preprocesor can be safely disabled if you are not using macros in your IDL file.

`rpcddsgen` application generates several files. Significant files to the developer are just a few and we will describe them in this section. The name of these files is generated using the interface's name defined in the IDL file. The *<InterfaceName>* nomenclature has to be substitute by the interface's name.

2.2.2 Server side

`rpcddsgen` generates a C++ source file with the definitions of the remote procedures and a C++ header file with the declaration of these remote procedures. These files are the skeleton of the servant that implements the defined interface and the developer can use each definition in the source file to implement the behavior of the remote procedure. These files are *<InterfaceName>*`ServerImpl.h` and *<InterfaceName>*`ServerImpl.cxx`.

Also `rpcddsgen` generates a C++ source file with an example of a server application and how create the server instance. This file is `Server.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 `Client.cxx`.

IMPORTANT: The IDL file name must be the same of the interface in order to compile the generated solution.

2.3 Implementation of the server

`rpcddsgen` application generates a class named `<InterfaceName>ServerImpl`. This class is a skeleton of a servant that implements the interface that the server will offer. This skeleton is located in the files `<InterfaceName>ServerImpl.h` and `<InterfaceName>ServerImpl.cxx`. All remote procedures are defined in this class, and the behavior of each one has to be implemented by the developer. For the remote procedure `deposit` in the IDL example in section *Example*, its definition is:

```
ReturnCode BankServerImpl::deposit(/*in*/const Account& ac, /*in*/ DDS_Long 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` contains a class that acts like the server. This class is implemented in files `<InterfaceName>Server.h` and `<InterfaceName>Server.cxx`. The class is named `<InterfaceName>Server` and it offers the interface implemented in the servant. A service's name is associated with this interface and client applications will use this service's name to connect with the server.

When an object of the class `<InterfaceName>Server` is created, proxies can establish a connection with it. How this connection is created and how the proxies found the server depends on the network transport that is set to be used by the server. These transports are described in section *Network transports*. By default servers use the UDP transport.

2.3.1 API

Using the suggested IDL example in section *Example*, the API of this class is:

```
class BankServer : public eProsima::RPCDDS::Server
{
    public:

        BankServer(std::string serviceName,
            eProsima::RPCDDS::ServerStrategy *strategy,
            int domainId = 0);

        BankServer(std::string serviceName ,
            eProsima::RPCDDS::ServerStrategy *strategy,
            eProsima::RPCDDS::Trnsport *transport, int domainId = 0);

        virtual ~BankServer();
```

The server provides two constructors. Both constructors expect in the `serviceName` parameter the service's name used by the server. In the `strategy` parameter is expected a server's strategy that defines how the server has to manage incoming requests. Server's strategies are described in the section *Threading Server strategies*. Also they permit to configure the DDS domain identifier with the `domainId` parameter.

The first constructor doesn't expect any more parameters and it creates a server that uses the UDP transport. The second constructor expects the network transport that will be used to establish connections with proxies.

2.3.2 Exceptions

In the server's side, developers can inform to the proxies about an error in the execution of the implemented remote procedures. eProsima RPC over DDS can catch the `eProsima::RPCDDS::ServerInternalException` exception in the developer's code. This exception will be delivered to the proxy and will be thrown in the proxy's side. An example of how this exception can be thrown:

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

   throw eProsima::RPCDDS::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;
eProsima::RPCDDS::ThreadPoolStrategy *pool = NULL;
BankServer *server = NULL;

try
{
   pool = new eProsima::RPCDDS::ThreadPoolStrategy(threadPoolSize);
   server = new BankServer("MyBankName", pool);
   server->serve();
}
catch(eProsima::RPCDDS::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

2.4 Implementation of the client

The code generated by `rpcddsgen` contains a class that acts like a proxy of the remote server. This class is implemented in files `<`*`InterfaceName`*`>Proxy.h` and `<`*`InterfaceName`*`>Proxy.cxx`. The proxy offers to the developer the server's interface and the developer can call its remote procedures directly.

The class is named `<`*`InterfaceName`*`>Proxy`. When an object of this class is created, a connection is established with the remote server. How this connection is created and how the server is found depends on the network transport that is set to be used by the proxy. These transports are described in section *Network transports*. By default proxies use the UDP transport.

2.4.1 API

Using the suggested IDL example in section *Example*, the API of this class is:

```
class BankProxy : public eProsima::RPCDDS::Client
{
    public:

        BankProxy(std::string remoteServiceName,
            int domainId = 0, long timeout = 10000);

        BankProxy(std::string remoteServiceName,
            eProsima::RPCDDS::Transport *transport,
            int domainId = 0, long timeout = 10000);

        virtual ~BankProxy();

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

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

};
```

The proxy provides two constructors. Both constructors expect in the `remoteServiceName` parameter the service's name used by the server to which the proxy wants to connect. Also they permit to configure the DDS domain identifier with the `domainId` parameter and through `timeout` parameter the maximum time for all remote procedure calls before the proxy returns a timeout exception.

The first constructor doesn't expect any more parameters and it creates a proxy that uses the UDP transport. The second constructor expects the network transport that will be used to establish the connection with the server.

The proxy provides to the developer the remote procedures. Using the suggested IDL in section *Example*, a 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 *Asynchronous calls*.

2.4.2 Exceptions

While a remote procedure call is execute, an error could occur. In these cases exceptions are used to report the error. Next exceptions can be thrown when a remote procedure is called:

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

All exceptions has the same base class `eProsima::RPCDDS::Exception`.

2.4.3 Example

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

```
BankProxy *proxy = NULL;

try
{
   proxy = new BankProxy(“MyBankName”);
}
catch(eProsima::RPCDDS::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}

Account ac;
DDS_Long  money ;
ReturnCode  depositRetValue;

Account_initialize(&ac);

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

3 Advanced concepts

3.1 Network transports

eProsima RPC over DDS provides three network transports. These transports define how a connection is established between a proxy and a server. The transports are:

- High performance and reliable UDP transport: The recommended option in LAN
- TCP transport: This transport is designed to be used in WAN cenarios.
- Shared memory transport: Enabled for same node communications.

3.1.1 UDP Transport

The purpose of this transport is to create a connection between a proxy and a server that are located in a local network. This transport is implemented by two classes. One is used by server's proxies and the other is used by servers.

3.1.1.1 UDPClientTransport

`UDPClientTransport` class implements a UDP transport that should be used by proxy's servers.

```
class UDPClientTransport : public Transport
{
   public:
      UDPClientTransport();
      UDPClientTransport(const char *to_connect);
      virtual ~UDPClientTransport();
};
```

This class has two constructors. The default constructor sets the UDP transport to utilize DDS discovery mechanism. This discovery mechanism allows to the proxy to find any server in the local network. There are two potential scenarios:

- In the local network there is only one server using the service's name requested. 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 remotely procedure, this server will execute this procedure and return the reply.
- In the local network there are several servers using the same service's name. This scenario could occur when the user wants to have redundant server 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.

When a proxy is created without a network transport, it creates internally a UDP transport and this transport is created with this constructor. The second constructor expects the IP address of the remote server in the `to_connect` parameter and then the proxy will connect with the server located in that IP address.

Using the suggested IDL example in the section *Example*, the developer could create a proxy that will connect with a specific server in a local network:

```
eProsima::RPCDDS::UDPClientTransport *udptransport = NULL;
BankProxy *proxy = NULL;

try
{
   udptransport = new eProsima::RPCDDS::UDPClientTransport(“192.168.1.12”);
   proxy = new BankProxy(“MyBankName”, udptransport);
}
catch(eProsima::RPCDDS::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}

Account ac;
DDS_Long  money ;
ReturnCode  depositRetValue;

Account_initialize(&ac);

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

3.1.1.2 UDPServerTransport

`UDPServerTransport` class implements a UDP transport that should be used by servers.

```
class UDPServerTransport : public Transport
{
   public:
      UDPServerTransport();
      virtual ~UDPServerTransport();
};
```

This class has one constructor. This constructor has no parameters and sets the UDP transport to utilize DDS discovery mechanism. DDS discovery mechanism allows to the server to discover any proxy in the local network. When a server is created without a network transport, it creates internally a UDP transport and this transport is created with this constructor.

Using the suggested IDL example in the section *Example*, the developer could create a server that will connect with any proxy in a local network:

```
unsigned int threadPoolSize = 5;
eProsima::RPCDDS::ThreadPoolStrategy *pool = NULL;
eProsima::RPCDDS::UDPServerTransport *udptransport = NULL;
BankServer *server = NULL;

try
{
   pool = new eProsima::RPCDDS::ThreadPoolStrategy(threadPoolSize);
   udptransport = new eProsima::RPCDDS::UDPServerTransport();
   server = new BankServer("MyBankName", pool, udptransport);

   server->serve();
}
catch(eProsima::RPCDDS::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

3.1.2 TCP Transport

The purpose of this transport is to create a connection between a proxy and a server that are located in a WAN. This transport is implemented by two classes. One is used by server's proxies and the other is used by servers.

3.1.2.1 TCPClientTransport

`TCPClientTransport` class implements a TPC transport that should be used by proxy's servers.

```
class TCPClientTransport : public Transport
{
   public:
      TCPClientTransport(const char *to_connect);
      virtual ~TCPClientTransport();
};
```

This class has one constructor. This constructor has one parameter. The parameter `to_connect` expects the public IP address and port of the remote server and then the proxy will connect with the server located in that public IP address. For more information see section *WAN communication*.

Using the suggested IDL example in the section *Example*, the developer could create a proxy that will connect with a server located in the public IP address `80.130.6.123` and port `7600`.

```
eProsima::RPCDDS::TCPClientTransport *tcptransport = NULL;
BankProxy *proxy = NULL;

try
{
   tcptransport = new
eProsima::RPCDDS::TCPClientTransport("80.130.6.123:7600");
   proxy = new BankProxy("MyBankName", tcptransport);
}
catch(eProsima::RPCDDS::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}

Account ac;
DDS_Long  money ;
ReturnCode  depositRetValue;

Account_initialize(&ac);

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

3.1.2.2 TCPServerTransport

`TCPServerTransport` class implements a TPC transport that should be used by servers.

```
class TCPServerTransport : public Transport
{
   public:
      TCPServerTransport(const char *public_address, const char *server_bind_port);
      virtual ~TCPServerTransport();
};
```

`This` class has one constructor. This constructor has two parameters. The parameter `public_address` expects the public IP address and port where a proxy could find the server. The parameter `server_bind_port` has to contain the local port that the server will open to make the connection. For more information see section *WAN communication*.

Using the suggested IDL example in the section *Example*, the developer could create a server that will be found in public IP address `80.130.6.123` and port `7600`. This server will open the port `7400` in its machine.

```
unsigned int threadPoolSize = 5;
eProsima::RPCDDS::ThreadPoolStrategy *pool = NULL;
eProsima::RPCDDS::TCPServerTransport *tcptransport = NULL;
BankServer *server = NULL;

try
{
   pool = new eProsima::RPCDDS::ThreadPoolStrategy(threadPoolSize);
   tcptransport = new eProsima::RPCDDS::TCPServerTransport("80.130.6.123:7600",
“7400");
   server = new BankServer(“MyBankName”, pool, tcptransport);

   server->serve();
}
catch(eProsima::RPCDDS::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 from a thread and the call won't 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 received as parameters the object that will be called when request had arrived and the input parameters of the remote procedure. Using the IDL example, `rpcddsgen` will generate next asynchronous method in the server proxy:

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

The asynchronous version of the remote procedures can generate an exception too.

The exceptions that could be thrown are:

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

Example:

```
class Bank_depositHandler : public depositCallbackHandler
{
    void deposit(/*out*/ ReturnCode deposit_ret)
    {
    }

    virtual void on_exception(const eProsima::RPCDDS::Exception &ex)
    {
    }
}

void main()
{
   BankProxy *proxy = NULL;

   try
   {
      proxy = new BankProxy("MyBankName");
   }
   catch(eProsima::RPCDDS::InitializeException &ex)
   {
      std::cout << ex.what() << std::endl;
   }

   Account ac;
   DDS_Long money  = 0;
   Bank_depositHandler deposit_handler;

   Account_initialize(&ac);

   try
   {
      proxy->deposit_async(deposit_handler, ac, money);
```

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 that 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 arrived. This function has as parameters the output parameters of the remote procedure. The other function is called in case of exception. User should create a class that inherits from *<InterfaceName>_<RemoteProcedureName>*`CallbackHandler` class and implement both abstract methods. Using the IDL example, `rpcddsgen` will generate next class:

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

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

Error code	Description
`eProsima::RPCDDS::ClientInternalException`	An exception occurs in the client side.
`eProsima::RPCDDS::ServerTimeoutException`	The maximum time was exceeded waiting the server’s reply.
`eProsima::RPCDDS::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 support one-way calls. A developer can define a remote procedure as one-way, and when the client application calls the remote procedure, the thread sends the request to the server but it won’t wait for the reply or an error.

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 output parameter. Any parameter cannot be defined with the reserved words `inout` or `out`.

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

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

3.4 Threading Server strategies

RPCDDS library offers several strategies that server could use when a request arrives. The subsection describes these strategies.

3.4.1 Single thread strategy

This is the simplest strategy. The server only uses one thread for request management. In this case the server only will be executing one request in time. The thread that server uses to manage the request is the reception thread of DDS. To use Single Thread Strategy, create the server providing a `SingleThreadStrategy` object in the constructor.

```
eProsima::RPCDDS::SingleThreadStrategy *single = NULL;
BankServer *server = NULL;

try
{
   single = new eProsima::RPCDDS::SingleThreadStrategy();
   server = new BankServer("MyBankName", single);

   server->serve();
}
catch(eProsima::RPCDDS::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. For each request arrived the server assigns the request to a free thread in the thread pool.

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

```
unsigned int threadPoolSize = 5;
eProsima::RPCDDS::ThreadPoolStrategy *pool = NULL;
BankServer *server = NULL;

try
{
   pool = new eProsima::RPCDDS::ThreadPoolStrategy(threadPoolSize);
   server = new BankServer("MyBankName", pool);

   server->serve();
}
catch(eProsima::RPCDDS::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 request arrived to processes the request.
To use Thread Pool Strategy, create the server providing a `ThreadPerRequestStrategy` object in the constructor.

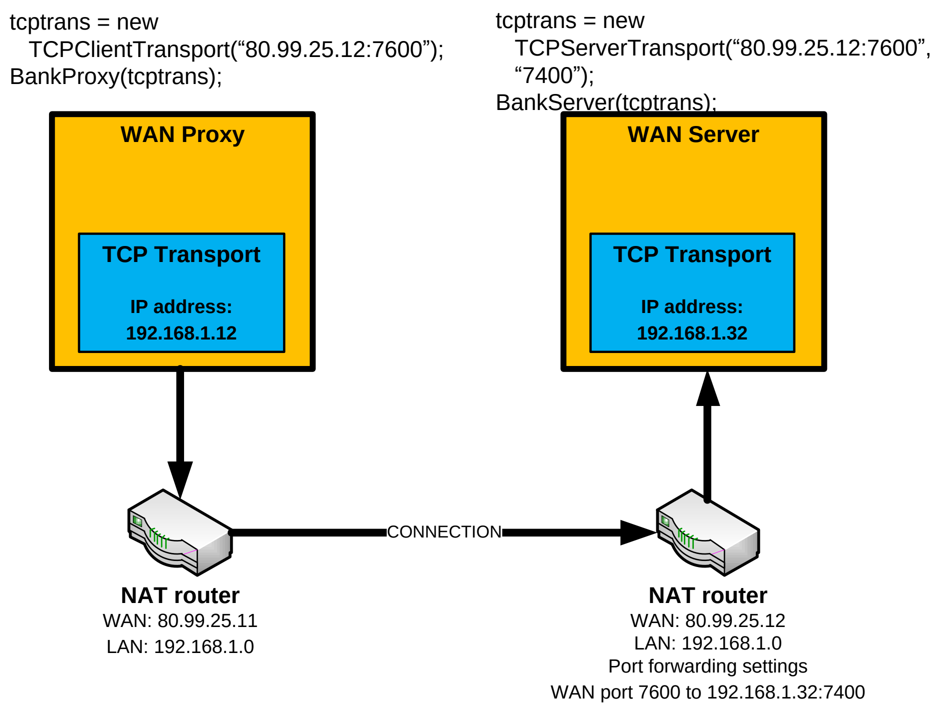
```
eProsima::RPCDDS::ThreadPerRequestStrategy *perRequest = NULL;
BankServer *server = NULL;

try
{
   perRequest = new eProsima::RPCDDS::ThreadPerRequestStrategy();
   server = new BankServer(“MyBankName”, perRequest);

   server->serve();
}
catch(eProsima::RPCDDS::InitializeException &ex)
{
   std::cout << ex.what() << std::endl;
}
```

4 WAN communication

eProsima RPC over DDS supports WAN networks through its TPC transport. A WAN server could be accessible at its public IP address and any WAN proxy could connect to this server. Usually a public server is behind a NAT with port forwarding. In this section is explained how to configure the network in this case.



The WAN server is located in a local network that has access to the WAN network through a NAT router. The local IP address of the computer where the WAN server will run is 192.168.1.32. It is decided that the WAN server will bind with the local port 7400 and it will be set with the parameter `server_bind_port` of the TCP transport. The public IP address of this NAT router is 80.99.25.12. It must be set a port forwarding configuration where data incoming in 7600 NAT router port will be forwarded to the local address 192.158.1.32 and local port 7400. Then the WAN server can be created with the `public_address` parameter of the TCP transport as `"80.99.25.12:7600"` and the `server_bind_port` parameter as `"7400"`.

WAN proxy could connect with this WAN server whether its public IP address and port is known. Then the WAN proxy can be created with the `to_connect` parameter of the TCP transport as `"80.99.25.12:7600"`.

5 Known Issues

5.1 RPCDDSGEN:

- RPCDDSGEN will not generate correct project files if the interface and the IDL file names are different.

6 HelloWorld example in Visual Studio 2010

In this section an example will be explain step by step. Only one remote procedure will be defined. A client will call this remote procedure, passing as parameter a string with a name. The server returns a new string that appends the name to a greeting sentence.

6.1 *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);
};
```

6.2 *Generating specific code*

Open a command prompt and go to the directory containing `HelloWorld.idl` file. Execute the following line:

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

6.3 *Implementation of the client*

Open the Visual Studio 2010 solution `HelloWorld-vs2010.sln`. `rpcddsgen` creates an example of a client application in the file `Client.cxx`. This example will use this base template. Two line will be added: one sets a value to the remote procedure parameter and the other prints the returned value in the output. Both lines are marked with a comment in the next example. Open the file `Client.cxx` and add it.

```
#include "HelloWorldProxy.h"
#include "HelloWorldRequestReplyPlugin.h"
#include “exceptions/Exceptions.h”

int main(int argc, char **argv)
{
    HelloWorldProxy *proxy = NULL;

    // Creation of the proxy for interface "HelloWorld".
    try
    {
        proxy = new HelloWorldProxy(“HelloService”);
    }
    catch(eProsima::RPCDDS::InitializeException &ex)
    {
        printf("Error: %s\n", ex.what()); // This line prints the error.
        return -1;
    }

    // Create and initialize parameters.
    char* name = strdup("Richard"); // This line set the remote procedure's
parameter.
    // Create and initialize return value.
    char* helloRetValue = NULL;

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

        printf("%s\n", helloRetValue); // This line prints the returned value.
    }
    catch(eProsima::RPCDDS::Exception &ex)
    {
        printf("Error: %s\n", ex.what()); // This line prints the error.
    }

    if(name != NULL) free(name);
    if(hello_ret != NULL) free(helloRetValue);

    delete(proxy);

    return 0;
}
```

6.4 Implementation of the server

`rpcddsgen` creates the server skelenton in the file `HelloWorldServerImpl.cxx`. In this file the remote procedure is defined and it has to be implemented. This example implements that the returned value will return a new string appending a greeting with the parameter of the remote procedure. Open the file and copy this behavior:

```
#include "HelloWorldServerImpl.h"

char* HelloWorldServerImpl::hello(/*in*/ char* name)
{
  char* hello_ret = NULL;

  // Allocate the returned value.
  hello_ret = (char*)calloc(100, 1);
  // Create the greeting sentence.
  sprintf(hello_ret, "Hello %s", name);

  return hello_ret;
}
```

6.5 Build and execute

Build the solution (F7) and go to `<`*`example_dir`*`>\objs\x64Win64VS2010` directory. Just double click on `HelloWorldServer.exe` to start the server. The server will inform that is running:

```
INFO<eProsima::RPCDDS::Server::Server>: Server is running
```

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

```
Hello Richard
```