RPCDDS

USER MANUAL

eProsima

Content

[Introduction 4](#_Toc341278514)

[Client/Server communications over RPCDDS 4](#_Toc341278515)

[RPCDDS 4](#_Toc341278516)

[Building a RPCDDS application 6](#_Toc341278517)

[Defining a set of remote procedures 6](#_Toc341278518)

[IDL Syntax and mapping to C++ 7](#_Toc341278519)

[Example 10](#_Toc341278520)

[Generating specific code 10](#_Toc341278521)

[Server side 11](#_Toc341278522)

[Client side 11](#_Toc341278523)

[Implementation of the server 12](#_Toc341278524)

[API 12](#_Toc341278525)

[Exceptions 13](#_Toc341278526)

[Example 13](#_Toc341278527)

[Implementation of the client 13](#_Toc341278528)

[API 13](#_Toc341278529)

[Exceptions 14](#_Toc341278530)

[Example 14](#_Toc341278531)

[Advanced concepts 16](#_Toc341278532)

[Network transports 16](#_Toc341278533)

[UDP Transport 16](#_Toc341278534)

[TCP Transport 17](#_Toc341278535)

[Asynchronous calls 18](#_Toc341278536)

[One-way calls 20](#_Toc341278537)

[Server strategies 20](#_Toc341278538)

[Single thread strategy 20](#_Toc341278539)

[Thread pool strategy 21](#_Toc341278540)

[Thread per request strategy 21](#_Toc341278541)

[WAN communication 21](#_Toc341278542)

[HelloWorld example in Visual Studio 2010 23](#_Toc341278543)

[Writing the IDL file 23](#_Toc341278544)

[Generating specific code 23](#_Toc341278545)

[Implementation of the client 23](#_Toc341278546)

[Implementation of the server 24](#_Toc341278547)

[Build and execute 25](#_Toc341278548)

# Introduction

The goal of this document is to introduce the RPCDDS product to developers and guide them in the implementation of the client/server paradigm in their distributed applications using RPCDDS.

## Client/Server communications over RPCDDS

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

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

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

RPCDDS provides an implementation of this general concept of invoking remote procedures. RPCDDS 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.



## RPCDDS

RPCDDS provides an easy way to invoke remote procedures using DDS standard as communication middleware. DDS (Data Distribution Service for Real-Time Systems) is an OMG specification of a data centric publish/subscribe communication model among real time software applications. RPCDDS comes with all benefits that DDS standard provides as reliable and efficient communications for distributed real time systems.

RPCDDS not only provide an easy way to export and invoke remote procedures. RPCDDS also brings other 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.
* RPCDDS provides several 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.
* RPCDDS supports several transports that DDS will use in the communications. There are two available transports. An UDP transport that brings the powerful benefit of DDS discovery in a local network or a TCP transport that allows connections with public servers located in internet.
* For DDS developers, RPCDDS allows enhancing DDS with client/service communications. A developer that uses DDS in its distributed application will be able to use a service-oriented interaction too.

# Building a RPCDDS application

RPCDDS 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 client calls these remote procedures should be transparent for the developer. From the point of view of the developer, an 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.

RPCDDS offers this transparency to the developer and facilitates the development. Using RPCDDS the developer only has to worry about:

* Define the set of remote procedures that server exposes and clients use.
* Generates the specific code for the defined set of remote procedures.
* Implement the client which calls the remote procedures as they have been defined.
* Implement the functionality of each remote procedure in the server.

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 .

## Defining a set of remote procedures

Interface Definition Language (IDL) is used by RPCDDS to define the remote procedures that server will offer to clients. Also type definitions used as parameters in these remote procedures are defined in the IDL file. The main IDL structure that RPCDDS supports is based in CORBA 2.x IDL and it is described in the following schema:

Procedure definitions

Interface definition

Type definitions

**IDL File**

RPCDDS comes with a java application named rpcddsgen. This application can read 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 .

As RPCDDS uses RTI DDS middleware in the communications, RPCDDS generates several types as RTI DDS types like in the case of simple types or sequences. Using RTI DDS types, it is easier to use both frameworks in the same application.

### IDL Syntax and mapping to C++

#### Simple types

RPCDDS 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.

Tabla : 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) \*/ |

#### 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. Some complex types cannot be used directly such as it will be indicated. The following table shows the supported complex types, how they are defined in the IDL file and what rpcddsgen generates in C++ language.

Table : 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.

#### 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 , 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.

#### 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);

rpcddsgenapplication maps the functions following these rules:

* The type of the returned value is mapped in C++ as it was described in section .
* 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 .

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

DDS\_Long funcName(DDS\_Short param1, DDS\_Long& param2);

#### 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.

#### 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 yet.

### 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.

## Generating specific code

Previous section shows how developers can define a set of remote procedures using IDL and also the types used in them. There is a way to generate specific code in C++ through these definitions in IDL. RPCDDS provides a tool for this purpose and this section describes it.

rpcddsgen is a Java application that reads an IDL file and generates the specific code in C++. With this generated code and the RPCDDS library a developer could implement the infrastructure to create clients and servers that supports these set of remote procedures.

RPCDDS provides a script to call the Java application rpcddsgendirectly. This script depends on the operation system. Next table shows the script that developer can use according to its operation system.

|  |  |
| --- | --- |
| **Operation System** | **Script’s name** |
| **Win 32-bits or 64-bits** | rpcddsgen.bat |
| **Linux 32-bits or 64-bits** | rpcddsgen.sh |

The way how this script should be call in a command line is:

rpcddsgen.bat [options] <IDL file>

This application accepts next options:

|  |  |
| --- | --- |
| **Option** | **Description** |
| **-language <language>** | Set the programming language used to generated the code. RPCDDS 1.0 only supports C++ language.  **Possible values:** C++  **Default value:** C++ |
| **-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 |

rpcddsgenapplication generates several files. Significant files to the developer are few and 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.

### Server side

rpcddsgengenerates 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.

### 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.

## 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 , 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 . By default servers use the UDP transport.

### API

Using the suggested IDL example in section , 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 . 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.

### Exceptions

In the server’s side, developers can inform to the proxies about an error in the execution of the implemented remote procedures. RPCDDS library 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;

}

### 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(“INGService”, pool);

server->serve();

}

catch(eProsima::RPCDDS::InitializeException &ex)

{

std::cout << ex.what() << std::endl;

}

## 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 . By default proxies use the UDP transport.

### API

Using the suggested IDL example in section , 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 , 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 .

### 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.

### Example

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

BankProxy \*proxy = NULL;

try

{

proxy = new BankProxy(“INGService”);

}

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;

}

# Advanced concepts

## Network transports

RPCDDS provides two network transports. These transports define how a connection is established between a proxy and a server. The transports are:

* UDP transport: This transport is designed to be used in local networks and uses the RTI DDS transport UDPv4.
* TCP transport: This transport is designed to be used in a WAN networks and it uses the RTI DDS transport TCPv4.

### UDP Transport

This transport is implemented by the class UDPTransport. Its purpose is to create a connection between a proxy and a server that are located in a local network.

class UDPTransport : public Transport

{

public:

/\* Constructor designed to proxies and servers \*/

UDPTransport();

/\* Constructor only designed to proxies \*/

UDPTransport(const char \*to\_connect);

virtual ~UDPTransport();

};

When a UDPTransport object is created, it can be employed in the proxy’s constructor or in the server’s constructor. But its behavior is different depending on which uses it.

#### Usage in a proxy

UDPTransport class has two constructors. Both constructors are applicable when a UDPTransport object will be used in the creation of a proxy. The default constructor sets the UDP transport to utilize RTPS discovery mechanism. RTPS discovery mechanism allows to the proxy to find any server in the local network. There are two scenarios that could be reached:

* 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 , the developer could create a proxy that will connect with a specific server in a local network:

eProsima::RPCDDS::UDPTransport \*udptransport = NULL;

BankProxy \*proxy = NULL;

try

{

udptransport = new eProsima::RPCDDS::UDPTransport(“192.168.1.12”);

proxy = new BankProxy(“INGService”, 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;

}

#### Usage in a server

UDPTransport class has two constructors but only one is applicable when a UDPTransport object will be used in the creation of a server. This constructor has no parameters and sets the UDP transport to utilize RTPS discovery mechanism. RTPS 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 , the developer could create a server that will connect with any proxy in a local network:

unsigned int threadPoolSize = 5;

eProsima::RPCDDS::ThreadPoolStrategy \*pool = new   
 eProsima::RPCDDS::ThreadPoolStrategy(threadPoolSize);

eProsima::RPCDDS::UDPTransport \*udptransport = new eProsima::RPCDDS::UDPTransport();

BankServer \*server = new BankServer(“INGService”, pool, udptransport);

server->serve();

### TCP Transport

This transport is implemented by the class TCPTransport. Its purpose is to create a connection between a proxy and a server that are located in a WAN network.

class TCPTransport : public Transport

{

public:

/\* Constructor only designed to servers \*/

TCPTransport(const char\* public\_address, const char\* server\_bind\_port);

/\* Constructor only designed to proxies \*/

TCPTransport(const char \*to\_connect);

virtual ~TCPTransport();

};

When a TCPTransport object is created, it can be employed in the proxy’s constructor or in the server’s constructor. But its behavior is different depending on which uses it.

#### Usage in a proxy

TCPTransport class has two constructors but only one is applicable when a TCPTransport object will be used in the creation of a proxy. 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 .

Using the suggested IDL example in the section , 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::TCPTransport \*tcptransport = new

eProsima::RPCDDS::TCPTransport("80.130.6.123:7600");

BankProxy \*proxy = new BankProxy(“INGService”, tcptransport);

Account ac;

DDS\_Long money ;

ReturnCode depositRetValue;

Account\_initialize(&ac);

depositRetValue = proxy->deposit(ac, money);

#### Usage in a server

TCPTransport class has two constructors but only one is applicable when a TCPTransport object will be used in the creation of a server. 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 .

Using the suggested IDL example in the section , 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 = new   
 eProsima::RPCDDS::ThreadPoolStrategy(threadPoolSize);

eProsima::RPCDDS::TCPTransport \*tcptransport = new   
 eProsima::RPCDDS::TCPTransport("80.130.6.123:7600", “7400");

BankServer \*server = new BankServer(“INGService”, pool, tcptransport);

server->serve();

## Asynchronous calls

Till now a client application could call a remote procedure from a thread. The call blocked the thread until the reply from the server had been received or an error had been occurred. RPCDDS library supports asynchronous calls. This means that a client application can call a remote procedure from a thread and this call won’t block the thread execution.

How is the client notified from the incoming reply? The client is notified through an object that the developer set in 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, rpcddsgenwill 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. |

How does client call an asynchronous invocation? 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. |

An example of how a client application should call an asynchronous invocation:

Class Bank\_depositHandler : public depositCallbackHandler

{

void deposit( /\*out\*/ ReturnCode deposit\_ret)

{

}

virtual void error(const eProsima::RPCDDS::Exception &ex)

{

}

}

BankProxy \*proxy = new BankProxy(“INGService”);

Account ac;

DDS\_Long money = 0;

Bank\_depositHandler deposit\_handler;

Account\_initialize(&ac);

proxy->deposit\_async(deposit\_handler, ac, money);

## One-way calls

Sometimes a remote procedure doesn’t need the reply from the server. For this cases, RPCDDS 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 onewayreserved 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);

};

## Server strategies

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

### 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 RTI DDS. An object from SingleThreadStrategy class must be passed to the server if user wants to activate this strategy.

eProsima::RPCDDS::SingleThreadStrategy \*single = new  
 eProsima::RPCDDS::SingleThreadStrategy();

BankServer \*server = new BankServer(“INGService”, single);

server->serve();

### Thread pool strategy

The server manages a thread pool that will be use to process the incoming requests. For each request arrived the server schedules the request for use a free thread in the thread pool. An object from ThreadPoolStrategy class must be passes to the server if user wants to activate this strategy.

unsigned int threadPoolSize = 5;

eProsima::RPCDDS::ThreadPoolStrategy \*pool = new  
 eProsima::RPCDDS::ThreadPoolStrategy(threadPoolSize);

BankServer \*server = new BankServer(“INGService”, pool);

server->serve();

### Thread per request strategy

For each new request arrived the server will create a new thread that processes the request. An object ThreadPerRequestStrategy class must be passes to the server if user wants to activate this strategy.

eProsima::RPCDDS::ThreadPerRequestStrategy \*perRequest = new  
 eProsima::RPCDDS::ThreadPerRequestStrategy();

BankServer \*server = new BankServer(“INGService”, perRequest);

server->serve();

## WAN communication

RPCDDS supports WAN networks through its TPC transport. A WAN server could be accessible in a 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 following figure will show the example:



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”.

# 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.

## 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);

};

## 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

## 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.cxxand add it.

/\*\*

\* Generated by RPCDDS \*

\* Example client. Method params should be initialized before execution \*

\*/

#include "HelloWorldProxy.h"

#include "HelloWorldRequestReplyPlugin.h"

#include “exceptions/Exceptions.h”

int main(int argc, char \*\*argv)

{

HelloWorldProxy \*proxy = new HelloWorldProxy(“HelloService”);

char\* name = strdup("Richard"); // This line set the remote procedure's parameter.

char\* helloRetValue = NULL;

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;

}

## 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:

/\*\*

\* Generated by RPCDDS \*

\* Empty interface implementation to be filled with your own code. \*

\*/

#include "HelloWorldServerImpl.h"

HelloWorldServerImpl::HelloWorldServerImpl()

{

}

HelloWorldServerImpl::~HelloWorldServerImpl()

{

}

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;

}

## Build and execute

Build the solution (F7) and go to <*example\_dir*>\objs\x64Win64VS2010directory. 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