DDSRPC

USER MANUAL

Contents

[1. Overview 2](#_Toc336605529)

[1.1. Communication patterns 2](#_Toc336605530)

[1.1.1. Publish/Subscribe 2](#_Toc336605531)

[1.1.2. Client/Server 2](#_Toc336605532)

[1.1.3. Peer to Peer 3](#_Toc336605533)

[1.2. Middleware Selection 3](#_Toc336605534)

[1.3. Client/Server Communications with DDS 3](#_Toc336605535)

[1.3.1. Generic Remote Procedure Call with DDS. 3](#_Toc336605536)

[2. Installation 4](#_Toc336605537)

[2.1. Windows 32-bits 4](#_Toc336605538)

[2.1.1. RTI DDS installation 4](#_Toc336605539)

[2.1.2. Boost C++ libraries installation 5](#_Toc336605540)

[2.1.3. DDSRPC installation 5](#_Toc336605541)

[2.1.1. Visual C++ 2010 Redistributable installation 6](#_Toc336605542)

[2.1. Windows 64-bits 6](#_Toc336605543)

[2.1.1. RTI DDS installation 6](#_Toc336605544)

[2.1.2. Boost C++ libraries installation 7](#_Toc336605545)

[2.1.3. DDSRPC installation 8](#_Toc336605546)

[2.1.4. Visual C++ 2010 Redistributable installation 8](#_Toc336605547)

[3. DDSRPC package content 8](#_Toc336605548)

[4. Basic concepts 9](#_Toc336605549)

[4.1. Defining a set of remote procedures 10](#_Toc336605550)

[4.1.1. IDL Syntax 11](#_Toc336605551)

[4.1.2. Example 15](#_Toc336605552)

[4.2. Generating specific code 15](#_Toc336605553)

[4.2.1. Other generated files 16](#_Toc336605554)

[4.3. Implementation of the client 18](#_Toc336605555)

[4.3.1. Local proxy 18](#_Toc336605556)

[4.3.2. WAN proxy 19](#_Toc336605557)

[4.4. Implementation of the server 20](#_Toc336605558)

[4.4.1. Local server 21](#_Toc336605559)

[4.4.1. WAN server 22](#_Toc336605560)

[5. Advanced concepts 22](#_Toc336605561)

[5.1. Asynchronous calls 22](#_Toc336605562)

[5.2. One-way calls 24](#_Toc336605563)

[5.3. Server strategies 24](#_Toc336605564)

[6. HelloWorld example in Visual Studio 2010 24](#_Toc336605565)

[6.1. Writing the IDL file 24](#_Toc336605566)

[6.2. Generating specific code 25](#_Toc336605567)

[6.3. Implementation of the client 26](#_Toc336605568)

[6.4. Implementation of the server 27](#_Toc336605569)

[6.5. Build and execute 28](#_Toc336605570)

# Overview

Distributed applications usually follow a communication pattern or paradigm to interact. The pattern selected must be the one best suited to the application functionality, where “best suited” is measured by the degree of commitment of several criteria as latency, throughput, bandwidth, hardware resources…

## Communication patterns

Actually there are three main patterns used in distributed software communications:

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

### Publish/Subscribe

Publish/subscribe (or pub/sub) is an asynchronous messaging paradigm where senders (publishers) of messages are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into classes, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology.

### Client/Server

Client-server computing or networking is a distributed application architecture that partitions tasks or work loads between service providers (servers) and service requesters, called clients. Often clients and servers operate over a computer network on separate hardware. A server machine is a high-performance host that is running one or more server programs which share its resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await (listen to) incoming requests.

### Peer to Peer

A peer-to-peer distributed network architecture is composed of participants that make a portion of their resources (such as processing power, disk storage or network bandwidth) directly available to other network participants, without the need for central coordination instances (such as servers or stable hosts). Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model where only servers supply, and clients consume.

## Middleware Selection

Middleware is software developed to carry out the common tasks related to communicate applications. This tasks are rather complex, and middleware is usually complex as well, but the aim of middleware is to hide that complexity from the rest of the application, making the development of distributed application as effortless as it would be if it wasn’t distributed.

To simplify development and use, middleware follows a communication pattern. So, as part of the designing a distributed application, a decision about what middleware to use must be made and it should be a middleware that implements a communication pattern suited to the application requirements.

Sadly, the best suited does not mean the best suited for all tasks. There are always some tasks that are best solved using other communication model. As a developer this ends up in you having to develop some complex “special cases” or worse: using a second middleware. That always increase the complexity of the application and development time (additional formation may be required) and resources consumption, which can be a hard problem to solve in real time systems.

## Client/Server Communications with DDS

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. It is a middleware that provides reliable and efficient communications for distributed real time systems.

To avoid the problems described previously, this client/server extension of DDS provides a generic solution to make calls to remote procedures over DDS infrastructure.

From an IDL definition of an interface, methods and types the extension generates a proxy to be called on client side and an empty skeleton to be filled with server functionality as you would get from any client/server middleware (CORBA, SOAP…).

### Generic Remote Procedure Call with DDS.

The following diagram describes the approach adopted by the DDS Client/Server extension.

For each method two topics are generated: one for the request and the other for the reply. The client writes requests instances on behalf of methods invocations in the proxy. On request reception, the server calls skeleton method and writes the result on the reply topic. The requests are completely identified, to be matched with the data received on the reply topic.



Figure 1: Generic Remote Procedure Invocation DDS style.

Although every developer skilled in DDS development should be able to do this himself, it takes time to develop it for each remote method/function and also may be hard to maintain.

# Installation

This section describes how to install DDSRPC in several operation systems. Select your operation system and follow the steps.

## Windows 32-bits

### RTI DDS installation

DDSRPC library uses RTI DDS as middleware communication and its libraries are necessary in the system. This DDSRPC library version has been compiled against RTI DDS 4.5f. Download the RTI DDS 4.5f installer and install the middleware.

|  |  |
| --- | --- |
| **OS** | **RTI DDS installation package** |
| **Windows 32-bits** | RTI\_Connext\_Messaging-4.5f-WIN32\_lic.exe |

The environment variable NDDSHOME needs to be set in your system. This environment variable has to point to the installation folder where RTI DDS 4.5f was installed. Also the environment variable PATH has to contain the location of the RTI DDS libraries for platform *i86Win32VS2010*. As example

|  |  |
| --- | --- |
| **Environment Variable** | **Value** |
| **NDDSHOME** | C:\Program Files (x86)\RTI\ndds.4.5f |
| **PATH** | %PATH%;%NDDSHOME%\lib\i86Win32VS2010 |

### Boost C++ libraries installation

DDSRPC library uses Boost C++ libraries and its libraries are necessary in the system. This DDSRPC version has been compiled against Boost C++ 1.51.0. Download and install the following packages:

* Against DDSRPC library with debug information:

|  |  |
| --- | --- |
| **Boost library** | **URL** |
| **Boost system** | [boost\_system-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_system-vc100-mt-gd-1_51.zip/download) |
| **Boost chrono** | [boost\_chrono-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_chrono-vc100-mt-gd-1_51.zip/download) |
| **Boost date time** | [boost\_date\_time-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_date_time-vc100-mt-gd-1_51.zip/download) |
| **Boost thread** | [boost\_thread-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_thread-vc100-mt-gd-1_51.zip/download) |

* Agains DDSRPC library without debug information:

|  |  |
| --- | --- |
| **Boost library** | **URL** |
| **Boost system** | [boost\_system-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_system-vc100-mt-1_51.zip/download) |
| **Boost chrono** | [boost\_chrono-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_chrono-vc100-mt-1_51.zip/download) |
| **Boost date time** | [boost\_date\_time-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_date_time-vc100-mt-1_51.zip/download) |
| **Boost thread** | [boost\_thread-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0/boost_thread-vc100-mt-1_51.zip/download) |

Boost C++ libraries have to be accessible for your system. Include in the environment variable PATH the location of the Boost C++ libraries.

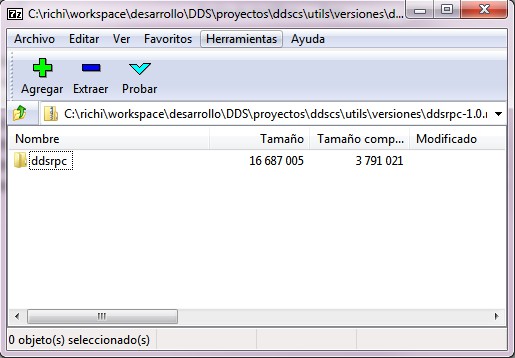
|  |  |
| --- | --- |
| **Environment Variable** | **Value** |
| **PATH** | %PATH%;C:\Program Files (x86)\boost\_1\_51\_0 |

### DDSRPC installation

Download the installation package for your system.

|  |  |  |
| --- | --- | --- |
| **OS** | | **DDSRPC installation package** |
| **Windows 32-bits** | ddsrpc-1.0.rc1-win\_RTIDDS-4.5f.zip | |

Open the zip file and extract the content over the RTI DDS 4.5f installation folder:



DDSRPC library has to be accessible for your system. Include in the environment variable PATH the location of the DDSRPC libraries for platform *i86Win32VS2010*.

|  |  |
| --- | --- |
| **Environment Variable** | **Value** |
| **PATH** | %PATH%;%NDDSHOME%\ddsrpc\lib\i86Win32VS2010 |

### Visual C++ 2010 Redistributable installation

Windows version of RTI DDS and DDSRPC library was developed using Visual C++ 2010 libraries. Usually systems don’t have Microsoft Visual Studio 2010 installed. In these cases the libraries has to be installed. Download and install the next package:

|  |  |  |
| --- | --- | --- |
| **OS** | | **DDSRPC installation package** |
| **Windows 32-bits** | [Visual C++ 2010 Redistributable](http://www.microsoft.com/en-us/download/details.aspx?id=8328) | |

## Windows 64-bits

### RTI DDS installation

DDSRPC library uses RTI DDS as middleware communication and its libraries are necessary in the system. This DDSRPC library version has been compiled against RTI DDS 4.5f. Download the RTI DDS 4.5f installer and install the middleware.

|  |  |
| --- | --- |
| **OS** | **RTI DDS installation package** |
| **Windows 64-bits** | RTI\_Connext\_Messaging-4.5f-WIN64\_lic.exe |

The environment variable NDDSHOME needs to be set in your system. This environment variable has to point to the installation folder where RTI DDS 4.5f was installed. Also the environment variable PATH has to contain the location of the RTI DDS libraries for platform *x64Win64VS2010*. As example

|  |  |
| --- | --- |
| **Environment Variable** | **Value** |
| **NDDSHOME** | C:\Program Files (x86)\RTI\ndds.4.5f |
| **PATH** | %PATH%;%NDDSHOME%\lib\x64Win64VS2010 |

### Boost C++ libraries installation

DDSRPC library uses Boost C++ libraries and its libraries are necessary in the system. This DDSRPC version has been compiled against Boost C++ 1.51.0. Download and install the following packages:

* Against DDSRPC library with debug information:

|  |  |
| --- | --- |
| **Boost library** | **URL** |
| **Boost system** | [boost\_system-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_system-vc100-mt-gd-1_51.zip/download) |
| **Boost chrono** | [boost\_chrono-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_chrono-vc100-mt-gd-1_51.zip/download) |
| **Boost date time** | [boost\_date\_time-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_date_time-vc100-mt-gd-1_51.zip/download) |
| **Boost thread** | [boost\_thread-vc100-mt-gd-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_thread-vc100-mt-gd-1_51.zip/download) |

* Agains DDSRPC library without debug information:

|  |  |
| --- | --- |
| **Boost library** | **URL** |
| **Boost system** | [boost\_system-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_system-vc100-mt-1_51.zip/download) |
| **Boost chrono** | [boost\_chrono-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_chrono-vc100-mt-1_51.zip/download) |
| **Boost date time** | [boost\_date\_time-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_date_time-vc100-mt-1_51.zip/download) |
| **Boost thread** | [boost\_thread-vc100-mt-1\_51.zip](http://sourceforge.net/projects/boost/files/boost-binaries/1.51.0-x64/boost_thread-vc100-mt-1_51.zip/download) |

Boost C++ libraries have to be accessible for your system. Include in the environment variable PATH the location of the Boost C++ libraries.

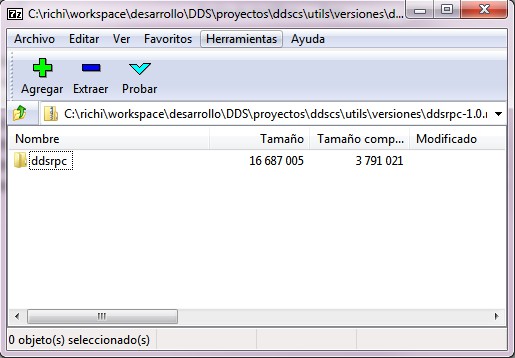
|  |  |
| --- | --- |
| **Environment Variable** | **Value** |
| **PATH** | %PATH%;C:\Program Files\boost\_1\_51\_0 |

### DDSRPC installation

Download the installation package for your system.

|  |  |  |
| --- | --- | --- |
| **OS** | | **DDSRPC installation package** |
| **Windows 64-bits** | ddsrpc-1.0.rc1-win\_RTIDDS-4.5f.zip | |

Open the zip file and extract the content over the RTI DDS 4.5f installation folder:



DDSRPC library have to be accessible for your system. Include in the environment variable PATH the location of the DDSRPC libraries for platform *x64Win64VS2010*.

|  |  |
| --- | --- |
| **Environment Variable** | **Value** |
| **PATH** | %PATH%;%NDDSHOME%\ddsrpc\lib\x64Win64VS2010 |

### Visual C++ 2010 Redistributable installation

Windows version of RTI DDS and DDSRPC library was developed using Visual C++ 2010 libraries. Usually systems don’t have Microsoft Visual Studio 2010 installed. In these cases the libraries has to be installed. Download and install the next package:

|  |  |  |
| --- | --- | --- |
| **OS** | | **DDSRPC installation package** |
| **Windows 64-bits** | [Visual C++ 2010 Redistributable](http://www.microsoft.com/en-us/download/details.aspx?id=13523) | |

# DDSRPC package content

This section explains the content of the DDSRPC installation folder.

#### Documentation

DDSRPC documentation is available in the directory *ddsrpc/doc*.

#### DDSRPC code generator

DDSRPC comes with a code generator. This application generates code according to the interface of remote procedures that user defines. The application was developed in Java language and the JAR files needed are located in the directory *ddsrpc/classes*.

The code generator can be executed easily with a script. This script is located in *ddsrpc/scripts* and its name is *ddsrpcgen.bat*.

#### DDSRPC library

DDSRPC library is distributed to support several operation systems. The following table shows where the DDSRPC library is according to the operation system:

|  |  |
| --- | --- |
| **OS** | **Location** |
| **Win 32-bits** | ddsrpc/lib/i86Win32VS2010 |
| **Win 64-bits** | ddsrpc/lib/x64Win64VS2010 |
| **Fedora 12 32-bits** |  |
| **Fedora 12 64-bits** |  |

#### Examples

DDSRPC comes with examples. The following table shows where the DDSRPC library is according to the operation system:

|  |  |
| --- | --- |
| **OS** | **Location** |
| **Win 32-bits** | ddsrpc/examples/i86Win32VS2010 |
| **Win 64-bits** | ddsrpc/ examples /x64Win64VS2010 |
| **Fedora 12 32-bits** |  |
| **Fedora 12 64-bits** |  |

# Basic concepts

DDSRPC allows a user to implement easily a distributed application using client/server paradigm. 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.

DDSRPC offers this transparency to the user and facilitates the development. Using DDSRPC the user 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.

This section will describe the basic concepts of these four steps that a user has to follow to implement its distributed application. Advanced concepts are described in next section.

## Defining a set of remote procedures

Interface Definition Language (IDL) is used in DDSRPC to define the remote procedures that server will offer to clients. Also type definitions used in the procedure’s parameters are defined using IDL too. The main IDL structure that DDSRPC supports is described in the following schema:

Procedure definitions

Interface definition

Type definitions

**IDL File**

DDSRPC come with a java application named *ddsrpcgen*. This application can read the IDL file and generates C++ code for the specific set of remote procedures that the user has defined. *ddsrpcgen* application will be described in the next subsection.

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

### IDL Syntax

#### Simple types

DDSRPC supports a variety of simple types that the user 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 it can be used in a definition of a complex type and what the *ddsrpcgen* generates in C++ language.

Tabla : Specifying Simple Types in IDL for C++

|  |  |  |
| --- | --- | --- |
| **IDL Type** | **Sample in IDL File** | **Sample Output Generated by ddsrpcgen** |
| **char** | struct PrimitiveStruct {  char char\_member; }; | typedef struct PrimitiveStruct {  DDS\_Char char\_member; } PrimitiveStruct; |
| **wchar** | struct PrimitiveStruct {  wchar wchar\_member; }; | typedef struct PrimitiveStruct {  DDS\_Wchar wchar\_member; } PrimitiveStruct; |
| **octet** | struct PrimitiveStruct {  octet octet\_member; }; | typedef struct PrimitiveStruct {  DDS\_Octet octet\_member; } PrimitiveStruct; |
| **short** | struct PrimitiveStruct {  short short\_member; }; | typedef struct PrimitiveStruct {  DDS\_Short short\_member; } PrimitiveStruct; |
| **unsigned short** | struct PrimitiveStruct {  unsigned short ushort\_member; }; | typedef struct PrimitiveStruct {  DDS\_UnsignedShort ushort\_member; }; |
| **long** | struct PrimitiveStruct {  long long\_member; }; | typedef struct PrimitiveStruct {  DDS\_Long long\_member; } PrimitiveStruct; |
| **unsigned long** | struct PrimitiveStruct {  unsigned long ulong\_member; }; | typedef struct PrimitiveStruct {  DDS\_UnsignedLong ulong\_member; }; |
| **long long** | struct PrimitiveStruct {  long long llong\_member; }; | typedef struct PrimitiveStruct {  DDS\_LongLong llong\_member; } PrimitiveStruct; |
| **unsigned long long** | struct PrimitiveStruct {  unsigned long long  ullong\_member; }; | typedef struct PrimitiveStruct {  DDS\_UnsignedLongLong  ullong\_member; } PrimitiveStruct; |
| **float** | struct PrimitiveStruct {  float float\_member; }; | typedef struct PrimitiveStruct {  DDS\_Float float\_member; } PrimitiveStruct; |
| **double** | struct PrimitiveStruct {  double double\_member; }; | typedef struct PrimitiveStruct {  DDS\_Double double\_member; } PrimitiveStruct; |
| **boolean** | struct PrimitiveStruct {  boolean boolean\_member; }; | typedef struct PrimitiveStruct {  DDS\_Boolean boolean\_member; } PrimitiveStruct; |
| **bounded string** | struct PrimitiveStruct {  string<20> string\_member; }; | typedef struct PrimitiveStruct {  char\* string\_member;  /\* maximum length = (20) \*/ } PrimitiveStruct; |
| **unbounded string** | struct PrimitiveStruct {  string string\_member; }; | typedef struct PrimitiveStruct {  char\* string\_member;  /\* maximum length = (2rr) \*/ } PrimitiveStruct; |
| **bounded wstring** | struct PrimitiveStruct {  wstring<20> wstring\_member; }; | typedef struct PrimitiveStruct {  DDS\_Wchar\* wstring\_member;  /\* maximum length = (20) \*/ } PrimitiveStruct; |
| **unbounded wstring** | struct PrimitiveStruct {  wstring wstring\_member; }; | typedef struct PrimitiveStruct {  DDS\_Wchar\* wstring\_member;  /\* maximum length = (255) \*/ } PrimitiveStruct; |

#### Complex types

Complex types can be created by the user using simple types defined previously. These complex types can be used as procedure’s parameters, although some complex types can be used directly. (See note below the next table). The following table shows the supported complex types, how it can be defined and what the *ddsrpcgen* generates in C++ language.

Tabla : Specifying Complex Types in IDL for C++

|  |  |  |
| --- | --- | --- |
| **IDL Type** | **Sample in IDL File** | **Sample Output Generated by ddsrpcgen** |
| **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)** | 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)** | struct SequenceStruct {  sequence<short,4>  short\_sequence; }; | typedef struct SequenceStruct {  DDSShortSeq short\_sequence; } SequenceStruct; |
| **unbounded sequence (See note)** | struct SequenceStruct {  sequence<short>  short\_sequence; }; | typedef struct SequenceStruct {  DDSShortSeq short\_sequence; } SequenceStruct; |

**Note:** Some 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 a 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. |

An example of how a procedure’s parameter should be defined inside a function is shown:

in long money

#### Function definition

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

* The returned value of the procedure. *void* type is allowed as returned value.
* 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 functionName(in short param1, inout 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 a interface should be defined is shown:

interface InterfaceExample

{

// Set of remote procedures.

};

#### Limitations

*ddsrpcgen* application has several limitations about IDL syntax:

* *ddsrpcgen* can handle just one interface per IDL file.
* Type definitions must be declared before the interface.
* The interface and the IDL must have the same name.
* Complex types (sequences, bounded strings, unions…) used in procedure definitions must be previously named using *typedef* keyword, as IDL 3.0 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);

};

## Generating specific code

As set of remote procedures has been defined by the user using the IDL syntax (as was described in previous section) and also type definitions used by these remote procedures, there should be a way to generate specific code for this defined stuff. DDSRPC provides a tool for this purpose. *ddsrpcgen* is a Java application that reads a IDL file and generates the specific code necessary for DDSRPC library. With this generated code and the DDSRPC library a user could implement the infrastructure to create clients and servers that supports these set of remote procedures.

To call the Java application *ddsrpcgen* directly by the user, DDSRPC provides an script. This script depends on the operation system. Next table shows the script that user can use according to its operation system.

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

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

**ddsrpcgen.bat [options] <IDL file>**

This application accepts next options:

|  |  |
| --- | --- |
| **Option** | **Description** |
| **-language <language>** | Set the programming language used to generated the code. DDSRPC 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 user to compile the client and the server.  **Possible values:** i86Win32VS2010, x64Win64VS2010, i86Linux2.6gcc4.4.3, x64Linux2.6gcc4.5.1 |

*ddsrpcgen* application generates several files. Significant files to the user are few and will describe in this section. For information about the rest of files, read the next subsection.

For the server side, *ddsrpcgen* generates a C++ source file with the definitions of the remote procedures and a C++ header file with the declaration of these remote procedures. User can use each definition in the source file to implement the behavior of the remote procedure. These files are:

|  |
| --- |
| **File Name** |
| **<InterfaceName>ServerImpl.h** |
| **<InterfaceName>SeverImpl.cxx** |

Also *ddsrpcgen* generates a C++ source file with an example of a server application and how create the server instance. This file is:

|  |
| --- |
| **File Name** |
| **Server.cxx** |

For the client side, *ddsrpcgen* generates a C++ source file with an example of a client application and how this client application can call a remote procedure. This file is:

|  |
| --- |
| **File Name** |
| **Client.cxx** |

### Other generated files

In this subsection the rest of generated files by *ddsrpcgen* are briefly described.

Tabla : Support for client implementation

|  |
| --- |
| **File Name** |
| **<InterfaceName>Proxy.h** |
| **<InterfaceName>Proxy.cxx** |
| **<InterfaceName>ClientRPCSupport.h** |
| **<InterfaceName>ClientRPCSupport.cxx** |
| **<InterfaceName>AsyncSupport.h** |
| **<InterfaceName>AsyncSupport.cxx** |

Tabla : Support for server implementation

|  |
| --- |
| **File Name** |
| **Server.cxx** |
| **<InterfaceName>Server.h** |
| **<InterfaceName>Server.cxx** |
| **<InterfaceName>ServerRPCSupport.h** |
| **<InterfaceName>ServerRPCSupport.cxx** |

Tabla : Support for types defined by user

|  |
| --- |
| **File Name** |
| **<InterfaceName>.h** |
| **<InterfaceName>.cxx** |
| **<InterfaceName>Plugin.h** |
| **<InterfaceName>Plugin.cxx** |
| **<InterfaceName>Support.h** |
| **<InterfaceName>Support.cxx** |

Tabla : Support for requests and replies

|  |
| --- |
| **File Name** |
| **<InterfaceName>RequestReply.h** |
| **<InterfaceName>RequestReply.cxx** |
| **<InterfaceName>RequestReplyPlugin.h** |
| **<InterfaceName>RequestReplyPlugin.cxx** |
| **<InterfaceName>RequestReplySupport.h** |
| **<InterfaceName>RequestReplySupport.cxx** |
| **<InterfaceName>RequestReplyUtils.h** |
| **<InterfaceName>RequestReplyUtils.cxx** |

Tabla : Visual 2010 solution

|  |
| --- |
| **File Name** |
| **<InterfaceName>-vs2010.sln** |
| **<InterfaceName>Server-vs2010.vcxproj** |
| **<InterfaceName>Client-vs2010.vcxproj** |

Tabla : Linux Makefile solution

|  |
| --- |
| **File Name** |
| **makefile\_<platform>** |

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

## Implementation of the client

For the examples the following interface definition in IDL is used:

interface Bank

{

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

};

The code generated by *ddsrpcgen* offers to the user several objects that act like proxy of the remote server. Their classes are implemented in files *<InterfaceName>Proxy.h* and *<InterfaceName>Proxy.cxx*. These proxies offer to the user the procedures that will call remotely.

### Local proxy

There is a class named *<InterfaceName>Proxy* that user can use to call remote procedures from a server. This class uses the RTI DDS transport UDPv4 to find the server using RTPS discovery. It only works in local networks. If there is one server or more then they will be found.

There are two scenarios that could be reached:

* In the local network there is only one server. When an object of the class *<InterfaceName>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. This scenario could occur when the user wants to have redundant server to avoid failures in the system. When an object of the class *<InterfaceName>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.

#### API

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

/\*\*

\* \brief This class implements a specific client's proxy for the defined interface by user.

\* This client's proxy uses the UDPv4 transport.

\*/

class BankProxy : public BankProxyH

{

public:

/\*\*

\* \brief Default constructor.

\*

\* \param domainId The DDS domain that DDS will use to work. Default value: 0

\* \param timeout Timeout used in each call to remotely procedures.

\* If the call exceeds the time, the call return a eProsima::DDSRPC::SERVER\_TIMEOUT.

\*/

BankProxy(int domainId = 0, long timeout = 10000);

virtual ~BankProxy();

};

#### Example

Using the suggested IDL example, the user can access to *desposit* procedure in the following way:

BankProxy \*proxy = new BankProxy();

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

DDS\_Long money ;

ReturnCode deposit\_ret ;

ReturnMessages depositRetValue ;

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

### WAN proxy

There is a class named *<InterfaceName>WANProxy* that user can use to call remote procedures from a server. This class uses the RTI DDS transport TCPv4 to find the server using RTPS discovery. It works in a WAN network. User specifies the public address and port of the server and RTI DDS will find the server and create a connection channel with it.

#### API

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

/\*\*

\* \brief This class implements a specific client's proxy for the defined interface by user.

\* This client's proxy uses the TCPv4 transport.

\*/

class BankWANProxy : public BankProxyH

{

public:

/\*\*

\* \brief Default constructor.

\*

\* \param to\_connect Public address and port for the server. By example: "218.18.3.133:7600"

\* \param domainId The DDS domain that DDS will use to work. Default value: 0

\* \param timeout Timeout used in each call to remotely procedures.

\* If the call exceeds the time, the call return a eProsima::DDSRPC::SERVER\_TIMEOUT.

\*/

BankWANProxy(const char \*to\_connect, int domainId = 0, long timeout = 10000);

virtual ~BankWANProxy();

};

#### Example

Using the suggested IDL example, the user can access to *desposit* procedure in the following way:

BankWANProxy \*proxy = new BankWANProxy("80.130.6.123:7600");

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

DDS\_Long money ;

ReturnCode deposit\_ret ;

ReturnMessages depositRetValue ;

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

## Implementation of the server

For the examples will be use the same interface definition in IDL as it was made in previous section.

interface Bank

{

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

};

*ddsrpcgen* application generates a class named *<InterfaceName>ServerImpl*. This class is a skeleton that contains all remote procedures server will offer and 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 user. For the remote procedure *deposit* in the IDL example, its definition is:

eProsima::DDSRPC::ReturnMessage

BankServerImpl::deposit(/\*in\*/const Account\* ac, const DDS\_Long money , /\*out\*/ReturnCode  
 &deposit\_ret)

{

eProsima::DDSRPC::ReturnMessage retCode = eProsima::DDSRPC::OPERATION\_SUCCESSFUL;

return retCode;

}

The code generated by *ddsrpcgen* offers to the user several objects that implement a server that offers the defined set of procedures. Their classes are implemented in files *<InterfaceName>Server.h* and *<InterfaceName>Server.cxx*.

### Local server

There is a class named *<InterfaceName>Server* that user can use to create a server in a local network. This class uses the RTI DDS transport UDPv4 to find clients using RTPS discovery. If a client is discovered then RTI DDS creates a connection channel with it.

#### API

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

/\*\*

\* \brief This class implements a specific server for the defined interface by user.

\* This server uses the UDPv4 transport.

\*/

class BankServer : public BankServerH

{

public:

/\*\*

\* \brief Default constructor.

\*

\* \param strategy Strategy used by server to work with new requests. Cannot be NULL.

\* \param domainId The DDS domain that DDS will use to work. Default value: 0

\*/

BankServer(eProsima::DDSRPC::ServerStrategy \*strategy,

int domainId = 0);

/// \brief The default destructor.

virtual ~BankServer();

};

#### Example

Using the suggested IDL example, the user can create the local server as:

unsigned int threadPoolSize = 5;

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

BankServer \*server = new BankServer(pool);

server->wait();

### WAN server

There is a class named *<InterfaceName>WANServer* that user can use to create a server that will be located after a public address. This class uses the RTI DDS transport TCPv4 to listen clients. If a client tries to connect then RTI DDS creates a connection channel with it.

#### API

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

/\*\*

\* \brief This class implements a specific server for the defined interface by user.

\* This server uses the TCPv4 transport.

\*/

class BankWANServer : public BankServerH

{

public:

/\*\*

\* \brief Default constructor.

\*

\* \param strategy Strategy used by server to work with new requests. Cannot be NULL.

\* \param public\_address Public address and port of the server. The server should be accesible in this address.

\* The user has to configure its router for this purpose. By example: "218.18.3.133:7600" \*

\* \param server\_bind\_port Port used by the server in its machine. This port will be use in the router for port forwarding

\* between the public port and this port.

\* \param domainId The DDS domain that DDS will use to work. Default value: 0

\*/

BankWANServer(eProsima::DDSRPC::ServerStrategy \*strategy,

const char \*public\_address, const char \*server\_bind\_port,

int domainId = 0);

/// \brief The default destructor.

virtual ~BankWANServer();

};

#### Example

Using the suggested IDL example, the user can create the WAN server as:

unsigned int threadPoolSize = 5;

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

BankWANServer \*server = new BankWANServer(pool, "80.130.6.123:7600", “7400");

server->wait();

# Advanced concepts

## Asynchronous calls

For the examples will be use the same interface definition in IDL as it was made in previous sections.

interface Bank

{

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

};

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. DDSRPC 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 user set in the asynchronous call. *ddsrpcgen* generates one class for each remote procedure that user will use in asynchronous calls. These classes are named *<InterfaceName>\_<RemoteProcedureName>*. Two methods are created inside these classes: one is called when the reply arrived and it has as parameters the output parameters of the remote procedure, the other is called in case of error. User should implement these two methods. Using the IDL example, *ddsrpcgen* will generate next class:

class Bank\_deposit

{

public:

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

{

}

virtual void error(eProsima::DDSRPC::ReturnMessage message)

{

}

};

How does client call an asynchronous invocation? *ddsrpcgen* 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, *ddsrpcgen* will generate next asynchronous method in the server proxy:

eProsima::DDSRPC::ReturnMessage

deposit\_async(Bank\_deposit &obj, /\*in\*/ const Account\* ac, const DDS\_Long money );

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

BankProxy \*proxy = new BankProxy();

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

DDS\_Long money = 0;

ReturnCode deposit\_ret;

eProsima::DDSRPC::ReturnMessage depositRetValue ;

Bank\_deposit deposit\_callbacks;

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

## One-way calls

For the examples will be use the next interface definition:

interface Bank

{

oneway void deposit(in Account ac, in long money);

};

Sometimes a remote procedure doesn’t need the reply from the server. For this cases, DDSRPC support one-way calls. A user 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.

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*.

## Server strategies

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

BankServer \*server = new BankServer(single);

server->wait();

### Thread pool strategy

The server manages a thread pool that will be use to process the incoming requests. For each request arrived the server schedule 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::DDSRPC::ThreadPoolStrategy \*pool = new eProsima::DDSRPC::ThreadPoolStrategy(threadPoolSize);

BankServer \*server = new BankServer(pool);

server->wait();

### 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::DDSRPC::ThreadPerRequestStrategy \*perRequest = new eProsima::DDSRPC:: ThreadPerRequestStrategy();

BankServer \*server = new BankServer(perRequest);

server->wait();

# 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:

*// HelloWorld.idl*

interface HelloWorld

{

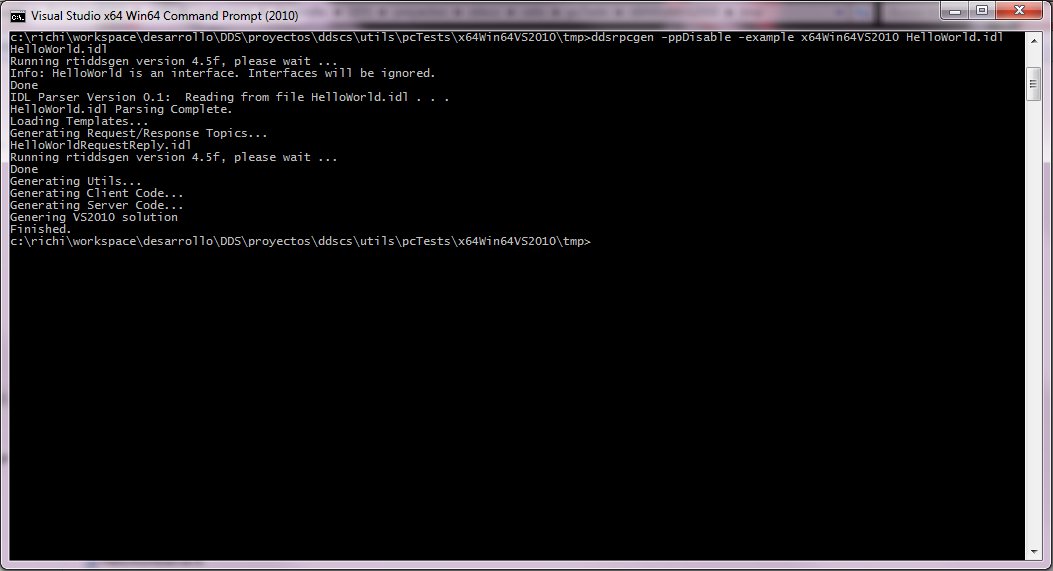
string hello(in string name);

};

## Generating specific code

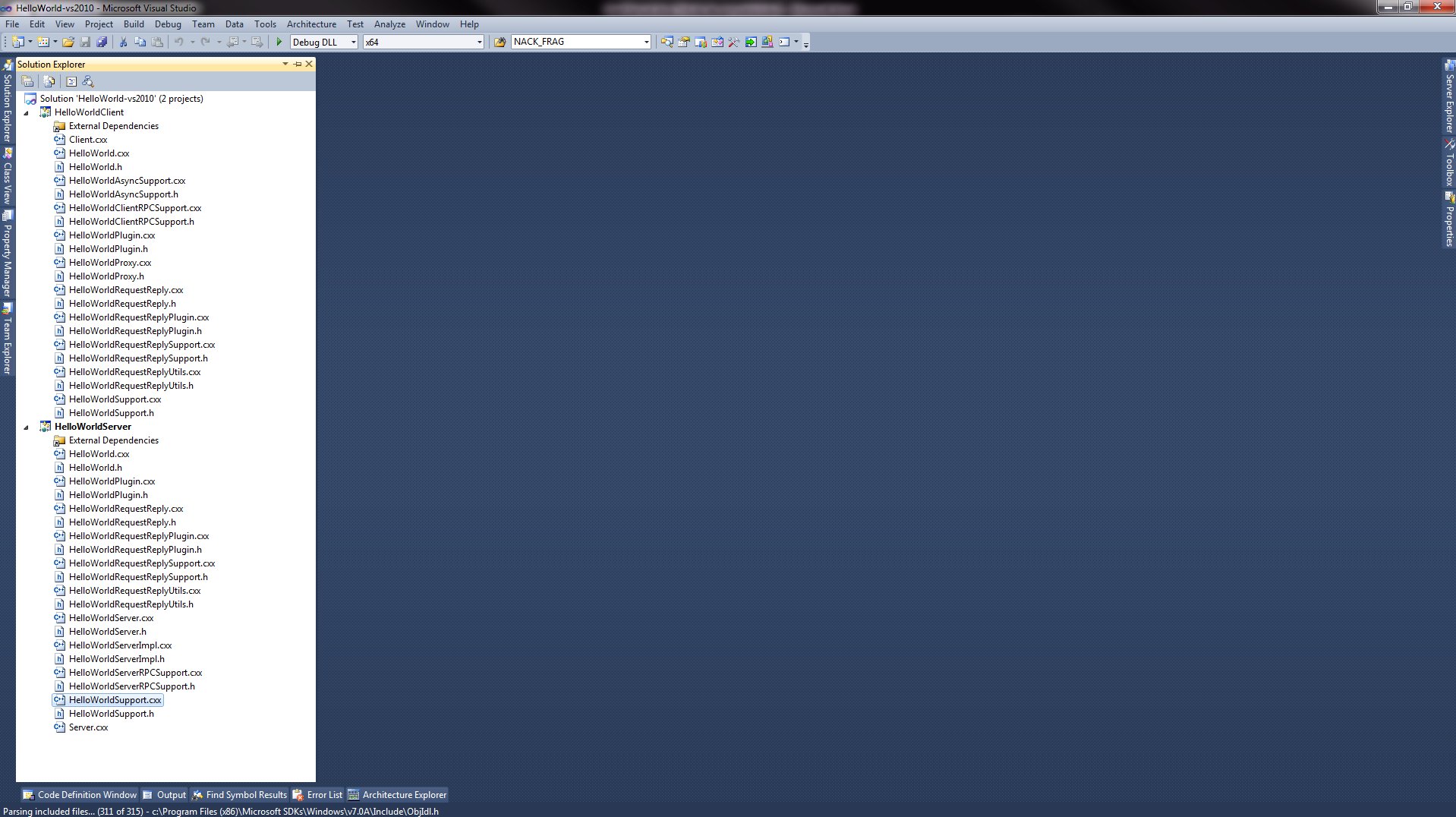
Open a command prompt and go to the directory containing HelloWorld.idl file. Execute the following line:

***ddsrpcgen -ppDisable -example x64Win64VS2010 HelloWorld.idl***



## Implementation of the client

Open the Visual Studio 2010 solution:



*ddsrpcgen* 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.

/\*\*

\* Generated by DDSRPC \*

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

\*/

#include "HelloWorldProxy.h"

#include "HelloWorldRequestReplyPlugin.h"

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

{

int domainId = 0;

unsigned int timeoutInMilliseconds = 10000;

HelloWorldProxy \*proxy = new HelloWorldProxy(domainId, timeoutInMilliseconds);

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

char\* hello\_ret = NULL;

eProsima::DDSRPC::ReturnMessage helloRetValue ;

helloRetValue = proxy->hello(name ,hello\_ret );

printf("%s\n", hello\_ret); // This line prints the returned value.

if(name != NULL) DDS::String\_free(name);

if(hello\_ret != NULL) DDS::String\_free(hello\_ret);

delete(proxy);

return 0;

}

## Implementation of the server

*ddsrpcgen* 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 DDSRPC \*

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

\*/

#include "HelloWorldServerImpl.h"

HelloWorldServerImpl::HelloWorldServerImpl()

{

}

HelloWorldServerImpl::~HelloWorldServerImpl()

{

}

eProsima::DDSRPC::ReturnMessage

HelloWorldServerImpl::hello(/\*in\*/ const char\* name , /\*out\*/ char\* &hello\_ret)

{

eProsima::DDSRPC::ReturnMessage retCode = eProsima::DDSRPC::OPERATION\_SUCCESSFUL;

// Allocate the returned value.

hello\_ret = (char\*)calloc(100, 1);

// Create the greeting sentence.

sprintf(hello\_ret, "Hello %s", name);

return retCode;

}

## 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 and then launch *HelloWorldClient.exe*. You will see:

