DDSRPC

USER MANUAL

eProsima

Contents

[Introduction 4](#_Toc338167732)

[Client/Server communications over DDSRPC 4](#_Toc338167733)

[DDSRPC 4](#_Toc338167734)

[Installation 6](#_Toc338167735)

[Windows 32-bits 6](#_Toc338167736)

[RTI DDS installation 6](#_Toc338167737)

[Boost C++ libraries installation 6](#_Toc338167738)

[DDSRPC installation 7](#_Toc338167739)

[Visual C++ 2010 Redistributable installation 7](#_Toc338167740)

[Windows 64-bits 7](#_Toc338167741)

[RTI DDS installation 7](#_Toc338167742)

[Boost C++ libraries installation 7](#_Toc338167743)

[DDSRPC installation 8](#_Toc338167744)

[Visual C++ 2010 Redistributable installation 8](#_Toc338167745)

[Building a DDSRPC application 10](#_Toc338167746)

[Defining a set of remote procedures 10](#_Toc338167747)

[IDL Syntax and mapping to C++ 11](#_Toc338167748)

[Example 17](#_Toc338167749)

[Generating specific code 17](#_Toc338167750)

[Implementation of the client 18](#_Toc338167751)

[Local proxy 18](#_Toc338167752)

[WAN proxy 20](#_Toc338167753)

[Implementation of the server 21](#_Toc338167754)

[Local server 21](#_Toc338167755)

[WAN server 22](#_Toc338167756)

[Advanced concepts 23](#_Toc338167757)

[Asynchronous calls 23](#_Toc338167758)

[One-way calls 24](#_Toc338167759)

[Server strategies 24](#_Toc338167760)

[Single thread strategy 24](#_Toc338167761)

[Thread pool strategy 24](#_Toc338167762)

[Thread per request strategy 25](#_Toc338167763)

[WAN communication 25](#_Toc338167764)

[HelloWorld example in Visual Studio 2010 27](#_Toc338167765)

[Writing the IDL file 27](#_Toc338167766)

[Generating specific code 27](#_Toc338167767)

[Implementation of the client 27](#_Toc338167768)

[Implementation of the server 28](#_Toc338167769)

[Build and execute 29](#_Toc338167770)

# Introduction

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

## Client/Server communications over DDSRPC

Distributed applications usually follow a communication pattern or paradigm to interact. 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).

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



## DDSRPC

DDSRPC 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. DDSRPC comes with all benefits that DDS standard provides as reliable and efficient communications for distributed real time systems.

DDSRPC not only provide an easy way to export and invoke remote procedures. DDSRPC also brings other features:

* Synchronous, asynchronous and one-way invocations. The synchronous invocation is the common invocation and it blocks the client 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 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.
* DDSRPC 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 a thread-pool’s threads to process the incoming requests. Thread-per-request strategy creates a new thread for each new incoming request. This new thread will process the request.
* DDSRPC 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 DDSRPC 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.

# 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.

**Windows 32-bits installer:** 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.

set NDDSHOME=C:\Program Files (x86)\RTI\ndds.4.5f

Also the environment variable PATH has to contain the location of the RTI DDS libraries for platform *i86Win32VS2010*. As example:

set 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.

set PATH=%PATH%;C:\Program Files (x86)\boost\_1\_51\_0

### DDSRPC installation

Download the installation package for your system.

**Windows 32-bits package:** 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*.

set 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:

**Visual C++ redistributable package (x86):** [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.

**Windows 64-bits installer:** 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.

set NDDSHOME=C:\Program Files (x86)\RTI\ndds.4.5f

Also the environment variable PATH has to contain the location of the RTI DDS libraries for platform *x64Win64VS2010*. As example:

set 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.

set PATH=%PATH%;C:\Program Files\boost\_1\_51\_0

### DDSRPC installation

Download the installation package for your system.

**Windows 64-bits package:** 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*.

set 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:

**Visual C++ redistributable package (x64):** [Visual C++ 2010 Redistributable](http://www.microsoft.com/en-us/download/details.aspx?id=13523)

# Building a DDSRPC application

DDSRPC 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.

DDSRPC offers this transparency to the developer and facilitates the development. Using DDSRPC 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 in DDSRPC to define the remote procedures that server will offer to clients. Also type definitions used in the procedure’s parameters are defined in the IDL file. 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 developer has defined. ddsrpcgen application will be described in the section .

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, it is easier to use both frameworks in the same application.

### IDL Syntax and mapping to C++

#### Simple types

DDSRPC 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 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 developer 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 ). The following table shows the supported complex types, how it can be defined and what the ddsrpcgen generates in C++ language.

Table : 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 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. |

Parameters that use the reserved word in are mapped in C++ as parameter-by-value. In other cases, when parameters use the reserved word inout and out, these parameters are mapped as parameter-by-reference. The following table shows the supported types, how it can be defined as parameter and what the ddsrpcgen generates in C++ language.

|  |  |  |
| --- | --- | --- |
| **IDL Type** | **Samples in IDL File** | **Sample Output Generated by ddsrpcgen** |
| **char** | in char char\_param  inout char char\_param  out char char\_param | /\*in\*/ const DDS\_Char char\_param  /\*inout\*/ DDS\_Char &char\_param  /\*out\*/ DDS\_Char &char\_param |
| **wchar** | in wchar wchar\_param  inout wchar wchar\_param  out wchar wchar\_param | /\*in\*/ const DDS\_Wchar wchar\_param  /\*inout\*/ DDS\_Wchar &wchar\_param  /\*out\*/ DDS\_Wchar &wchar\_param |
| **octet** | in octet octet\_param  inout octet octet\_param  out octet octet\_param | /\*in\*/ const DDS\_Octet octet\_param  /\*inout\*/ DDS\_Octet &octet\_param  /\*out\*/ DDS\_Octet &octet\_param |
| **short** | in short short\_param  inout short short\_param  out short short\_param | /\*in\*/ const DDS\_Short short\_param  /\*inout\*/ DDS\_Short &short\_param  /\*out\*/ DDS\_Short &short\_param |
| **unsigned short** | in unsigned short ushort\_param  inout unsigned short ushort\_param  out unsigned short ushort\_param | /\*în\*/ const DDS\_UnsignedShort ushort\_param  /\*inout\*/ DDS\_UnsignedShort &ushort\_param  /\*out\*/ DDS\_UnsignedShort &ushort\_param |
| **long** | in long long\_param  inout long long\_param  out long long\_param | /\*in\*/ const DDS\_Long long\_param  /\*inout\*/ DDS\_Long &long\_param  /\*out\*/ DDS\_Long &long\_param |
| **unsigned long** | in unsigned long ulong\_param  inout unsigned long ulong\_param  out unsigned long ulong\_param | /\*in\*/ const DDS\_UnsignedLong ulong\_param  /\*inout\*/ DDS\_UnsignedLong &ulong\_param  /\*out\*/ DDS\_UnsignedLong &ulong\_param |
| **long long** | in long long llong\_param  inout long long llong\_param  out long long llong\_param | /\*in\*/ const DDS\_LongLong llong\_param  /\*inout\*/ DDS\_LongLong &llong\_param  /\*out\*/ DDS\_LongLong &llong\_param |
| **unsigned long long** | in unsigned long long  ullong\_param  inout unsigned long long  ullong\_param  out unsigned long long  ullong\_param | /\*in\*/ const DDS\_UnsignedLongLong  ullong\_param  /\*inout\*/ DDS\_UnsignedLongLong  &ullong\_param  /\*out\*/ DDS\_UnsignedLongLong  &ullong\_param |
| **float** | in float float\_param  inout float float\_param  out float float\_param | /\*in\*/ const DDS\_Float float\_param  /\*inout\*/ DDS\_Float &float\_param  /\*out\*/ DDS\_Float &float\_param |
| **double** | in double double\_param  inout double double\_param  out double double\_param | /\*in\*/ const DDS\_Double double\_param  /\*inout\*/ DDS\_Double &double\_param  /\*out\*/ DDS\_Double &double\_param |
| **boolean** | in boolean boolean\_param  inout boolean boolean\_param  out boolean boolean\_param | /\*in\*/ const DDS\_Boolean boolean\_param  /\*inout\*/ DDS\_Boolean boolean\_param  /\*out\*/ DDS\_Boolean boolean\_param |
| **bounded string** | in string<20> string\_param  inout string<20> string\_param  out string<20> string\_param | /\*in\*/ const char\* string\_param  /\*inout\*/ char\* &string\_param  /\*out\*/ char\* &string\_param |
| **unbounded string** | in string string\_param  inout string string\_param  out string string\_param | /\*in\*/ const char\* string\_param  /\*inout\*/ char\* &string\_param  /\*out\*/ char\* &string\_param |
| **bounded wstring** | in wstring<20> wstring\_param  inout wstring<20> wstring\_param  out wstring<20> wstring\_param | /\*in\*/ const DDS\_Wchar\* wstring\_param;  /\*inout\*/ DDS\_Wchar\* &wstring\_param  /\*out\*/ DDS\_Wchar\* &wstring\_param |
| **unbounded wstring** | in wstring wstring\_param  inout wstring wstring\_param  out wstring wstring\_param | /\*in\*/ const DDS\_Wchar\* wstring\_param  /\*in\*/ DDS\_Wchar\* &wstring\_param  /\*in\*/ DDS\_Wchar\* &wstring\_param |
| **enum** | in PrimitiveEnum enum\_param  inout PrimitiveEnum enum\_param  out PrimitiveEnum enum\_param | /\*in\*/ const PrimitiveEnum enum\_param  /\*inout\*/ PrimitiveEnum &enum\_param  /\*out\*/ PrimitiveEnum &enum\_param |
| **struct** | in PrimitiveStruct struct\_param  inout PrimitiveStruct struct\_param  out PrimitiveStruct struct\_param | /\*in\*/ const PrimitiveStruct\* struct\_param  /\*inout\*/ PrimitiveStruct\* &struct\_param  /\*out\*/ PrimitiveStruct\* &struct\_param |
| **union** | in PrimitiveUnion union\_param  inout PrimitiveUnion union\_param  out PrimitiveUnion union\_param | /\*in\*/ const PrimitiveUnion\* union\_param  /\*inout\*/ PrimitiveUnion\* &union\_param  /\*out\*/ PrimitiveUnion\* &union\_param |
| **typedef** | in TypedefShort typedef\_param  inout TypedefShort typedef\_param  out TypedefShort typedef\_param | /\*in\*/ TypedefShort typedef\_param  /\*inout\*/ TypedefShort &typedef\_param  /\*out\*/ TypedefShort &typedef\_param |

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 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 funcName(in short param1, inout long param2);

ddsrpcgenapplication maps the functions following these rules:

* 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.
* The returned value in the defined function in IDL is considered as a parameter in the C++ function. This parameter is located at the end of all parameters and it is an output parameter. In the case that the returned value in the defined function is the void type, then it is not added any parameter at the end of the C++ function.
* The returned value of the C++ function is always of type eProsima::DDSRPC::ReturnMessage. This type is an enumerator that returns the success or the error in the operation.

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

eProsima::DDSRPC::ReturnMessage  
funcName(const DDS\_Short param1,const DDS\_Long param2, DDS\_Long &funcName\_ret);

#### 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. DDSRPC 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

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 (array and sequences) 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 developer 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 developer could implement the infrastructure to create clients and servers that supports these set of remote procedures.

To call the Java application ddsrpcgendirectly by the developer, DDSRPC provides a script. 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** | 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 developer to compile the client and the server.  **Possible values:** i86Win32VS2010, x64Win64VS2010, i86Linux2.6gcc4.4.3, x64Linux2.6gcc4.5.1 |

ddsrpcgenapplication 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

ddsrpcgengenerates 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 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  
 <*InterfaceName*>ServerImpl.cxx

Also ddsrpcgen 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

ddsrpcgen 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 client

For the examples in this section 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 developer several classes that act like proxies of the remote server. Their differences are about what network transport they use. These classes are implemented in files <*InterfaceName*>Proxy.h and <*InterfaceName*>Proxy.cxx. These proxies offer to the developer the remote procedures of the remote server and the developer can call them directly using these proxies.

### Local proxy

There is a class named <*InterfaceName*>Proxy that the developer can use to call remote procedures from a server located in the local network. This class uses the RTI DDS transport UDPv4. It can find the server using RTPS discovery or the developer can set the address of the server directly. It only works in local networks. If this local proxy is set to use RTPS discovery and there is one server or more then they will be found. In this case 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:

class BankProxyH : public eProsima::DDSRPC::Client

{

public:

eProsima::DDSRPC::ReturnMessage deposit(/\*in\*/ const Account\* ac, /\*in\*/ const DDS\_Long money, /\*out\*/ ReturnCode &deposit\_ret);

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

};

class BankProxy : public BankProxyH

{

public:

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

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

virtual ~BankProxy();

};

The local proxy provides two constructors. Both constructors permit to configure the DDS domain identifier with the domainId parameter and through timeout parameter the maximum time for all remote procedures call before the local proxy returns a timeout error. The first constructor doesn’t expect any more parameters and it creates a local proxy that use RTPS discovery to find the server. The second constructor expects the IP address of the remote server in the to\_connect parameter and it creates a local proxy that will connect with the server located in that IP address.

The local proxy provides to the developer the remote procedures. Using the suggested IDL the local 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 .

#### Example

Using the suggested IDL example, the developer can access to deposit 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 located in a WAN network. 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. For more information see section .

#### API

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

class BankProxyH : public eProsima::DDSRPC::Client

{

public:

eProsima::DDSRPC::ReturnMessage deposit(/\*in\*/ const Account\* ac, /\*in\*/ const DDS\_Long money, /\*out\*/ ReturnCode &deposit\_ret);

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

};

class BankWANProxy : public BankProxyH

{

public:

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

virtual ~BankWANProxy();

};

The WAN proxy provides one constructor. This constructor permits to configure the DDS domain identifier with the domainId parameter and through timeout parameter the maximum time for all remote procedures call before the WAN proxy returns a timeout error. This constructor expects the public IP address and port of the remote server in the to\_connect parameter and it creates a WAN proxy that will connect with the server located in that public IP address.

The WAN proxy provides to the developer the remote procedures. Using the suggested IDL the WAN 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 .

#### Example

Using the suggested IDL example, the user can access to deposit 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 in this section 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 developer. 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 developer several classes that implement a server. Their differences are about what network transport they use. These classes are implemented in files <*InterfaceName*>Server.h and <*InterfaceName*>Server.cxx.

### Local server

There is a class named <*InterfaceName*>Server that the developer 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:

class BankServer : public BankServerH

{

public:

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

int domainId = 0);

virtual ~BankServer();

};

The local server provides one constructor. This constructor permits to configure the DDS domain identifier with the domainId. This constructor expects in the strategy parameter a server’s strategy that defines how the server has to manage incoming requests. Server’s strategies are described in the section .

#### 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 the developer 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:

class BankWANServer : public BankServerH

{

public:

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

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

int domainId = 0);

virtual ~BankWANServer();

};

The WAN server provides one constructor. This constructor permits to configure the DDS domain identifier with the domainId. This constructor expects in the strategy parameter a server’s strategy that defines how the server has to manage incoming requests. Server’s strategies are described in the section . Also the constructor expects the public IP address and port in the public\_address parameter when client could find it and in the server\_bind\_port parameter the local port that the WAN server will open. For more information see section .

#### 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 in this section 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 developer 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, ddsrpcgenwill 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 an 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 in this section 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 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.

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.

## 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 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::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();

## WAN communication

DDSRPC supports WAN networks. 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. 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 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 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:

ddsrpcgen -ppDisable -example x64Win64VS2010 HelloWorld.idl

## Implementation of the client

Open the Visual Studio 2010 solution HelloWorld-vs2010.sln. 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.cxxand 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\x64Win64VS2010directory. Just double click on HelloWorldServer.exe to start the server. The server will inform that is running:

INFO<eProsima::DDSRPC::Server::Server>: Server is running

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

Hello Richard