USER MANUAL

DDS Client/Server

Contents

[1. Overview 2](#_Toc327462367)

[1.1. Communication patterns 2](#_Toc327462368)

[1.1.1. Publish/Subscribe 2](#_Toc327462369)

[1.1.2. Client/Server 2](#_Toc327462370)

[1.1.3. Peer to Peer 2](#_Toc327462371)

[1.2. Middleware Selection 2](#_Toc327462372)

[1.3. Client/Server Communications with DDS 3](#_Toc327462373)

[1.3.1. Generic Remote Procedure Call with DDS. 3](#_Toc327462374)

[2. Installation 4](#_Toc327462375)

[2.1. Installing RTI DDS 4](#_Toc327462376)

[2.2. Installing DDS Client/Server Extension 9](#_Toc327462377)

[2.3. DDSCS Runtime Libraries 9](#_Toc327462378)

[2.4. Microsoft Runtime Libraries 9](#_Toc327462379)

[3. Usage 10](#_Toc327462380)

[3.1. IDL 10](#_Toc327462381)

[3.1.1. QoS profile definitions 10](#_Toc327462382)

[3.1.2. Example 11](#_Toc327462383)

[3.2. Code generation 12](#_Toc327462384)

[3.2.1. Output 12](#_Toc327462385)

[3.3. Client 13](#_Toc327462386)

[3.4. Server 14](#_Toc327462387)

[3.5. Suggested QoS 15](#_Toc327462388)

[4. Visual Studio HelloWorld Example 15](#_Toc327462389)

[4.1. Write an IDL File. 15](#_Toc327462390)

[4.2. Execute code generator 15](#_Toc327462391)

[4.3. Open VS2005 Solution 17](#_Toc327462392)

[4.4. Write your Code 18](#_Toc327462393)

[4.5. Build and execute 20](#_Toc327462394)

# 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, the 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.



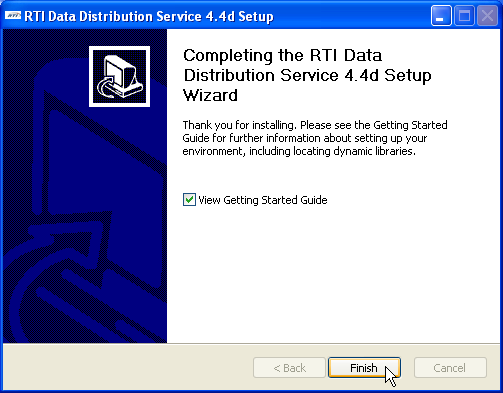
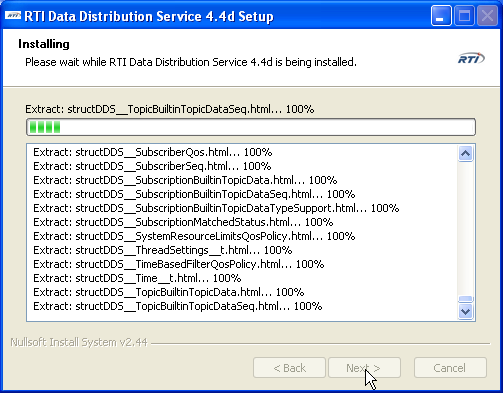
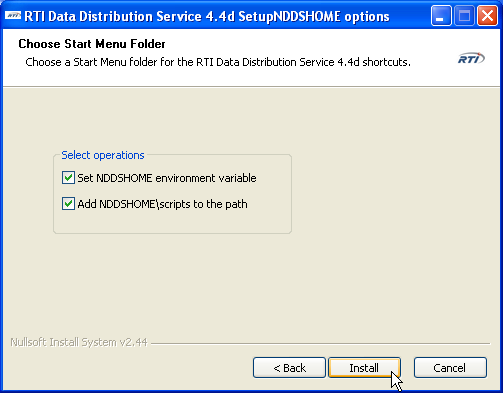
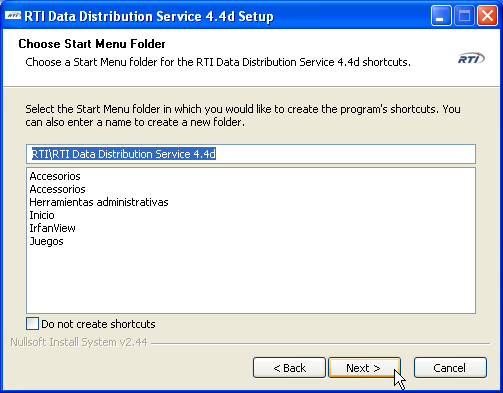
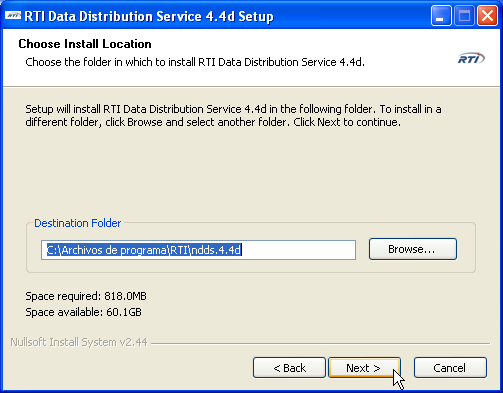
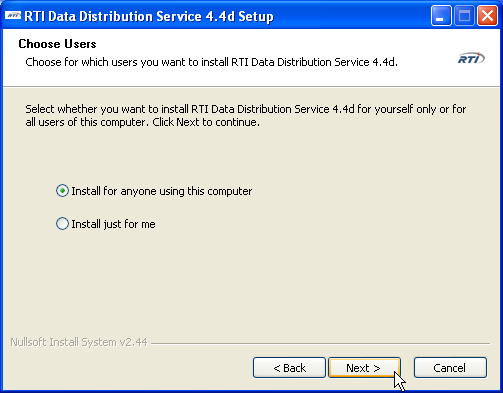
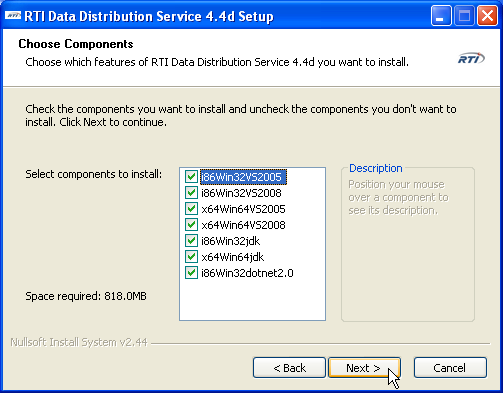
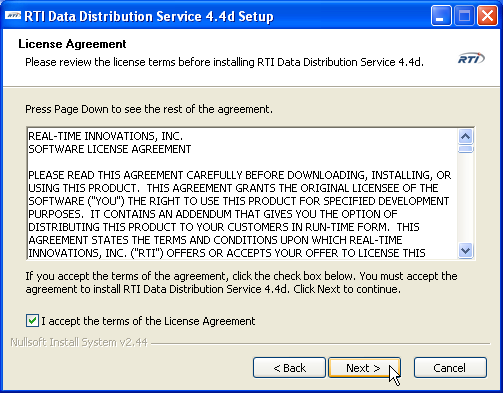
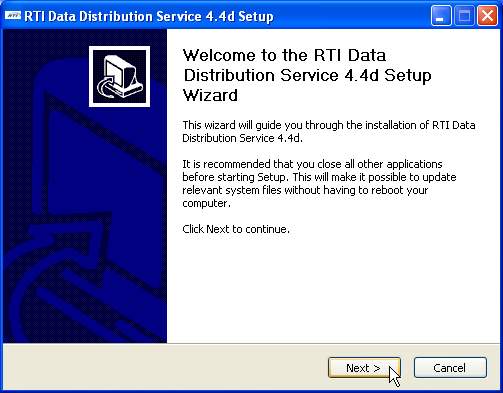
Figure 1: Generic Remote Procedure Invocation DDS style.

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

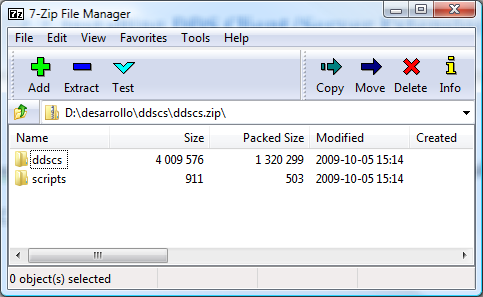
## Installing RTI DDS

First step before using the DDS Client/Server is the installation of the RTI DDS 4.5f middleware. Execute the installer *rtidds45f-Windows-installer.exe* and follow the installer’s steps.



## Installing DDS Client/Server Extension

Second step is installing the DDS Client/Server extension. Open the zip file ddscs-1.0\_RTIDDS-4.5f*.zip*



and extract it over the RTI DDS installation folder:

*C:/Program Files/RTI/ndds.4.5f*

If you checked the “Add NDDSHOME\scripts to the path” box installing RTI DDS, you can call ddscs.bat from any directory.

## DDSCS Runtime Libraries

In case of generating a dynamically linked application *ddscsd.dll* or *ddscs.dll* must be on the $PATH environment variable or be copied to the same directory than your application executable. They can be found at $NDDSHOME\ddscs\lib\i86Win32VS2005 directory.

## Microsoft Runtime Libraries

Unfortunately, Microsoft runtime libraries updates have compatibility problems. The tool is linked against last “important” security patch of 07-06-2011 and, consequently, the Visual Studio Redistributable package of 07-06-2011is required. It can be found in:

Visual 2005:

<http://www.microsoft.com/es-es/download/details.aspx?id=26347>

Visual 2008:

<http://www.microsoft.com/es-es/download/details.aspx?id=26368>

In order to debug your application, you may also need to update Visual Studio.

# Usage

## IDL

Interface Definition Language (IDL) is used to define the procedures that server will offer to clients. Also type definitions, used in procedures, are defined too. The main IDL structure that DDS Client/Server supports is described in the following schema:

Procedure definitions

Interface definition

Type definitions

**IDL File**

DDS Client/Server comes with a code generator that parses IDL files to generate the client’s proxy and the server’s skeleton. This generator has some limitations:

* It 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.

### QoS profile definitions

DDS Client/Server uses DDS entities in its infrastructure. IDL is transformed in code by the code generator. Some IDL’s definitions are transformed to classes that use some DDS entities. The relationship between IDL’s definitions and DDS entities are shown in next table. This matter will be explained with more details in next sections.

|  |  |  |
| --- | --- | --- |
| **IDL definition** | **Classes generated** | **DDS entities used** |
| **Interface** | InterfaceServer, InterfaceProxy | DomainParticipant |
| **Operation** | OperationServerRemoteService, OperationClientRemoteService | DataReader, DataWriter |

DDS entities support configuration over Quality of Service (QoS). This QoS can be stored in a default XML file named *USER\_QOS\_PROFILES.xml*. In this XML file, Qos is structured in libraries that contain QoS profiles. This feature can be used in DDS Client/Service and IDL is used to specify the QoS library and QoS profile for each operation and the interface. The IDL systax that sets the QoS for an interface is:

|  |  |  |
| --- | --- | --- |
| **IDL syntax** | **Description** | **Example** |
| **//#qos** | Set the QoS library and QoS profile to the DomainParticipant. | interface Bank  {  }; //#qos QoSLibrary QoSProfile |

The IDL syntax that sets the QoS for an operation is:

|  |  |  |
| --- | --- | --- |
| **IDL syntax** | **Description** | **Example** |
| **//#qos** | Set the QoS library and QoS profile to both DataWriter and Datareader. | //#qos QoSLibrary QoSProfile in op(in long param); |
| **//#qos-request** | Set the QoS library and QoS profile to the entity (DataWriter or DataReader) that is used in request. | //#qos-request QoSLibrary QoSProfile in op(in long param); |
| **//#qos-reply** | Set the QoS library and QoS profile to the entity (DataWriter or DataReader) that is used in reply. | //#qos-reply QoSLibrary QoSProfile in op(in long param); |

### Example

A simple example of IDL structure is shown:

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);

}; //#qos Bank\_Library Bank\_Profile

## Code generation

To generate a client and server applications you need to call ddcs script from command line.

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

This application accepts some arguments.

|  |  |
| --- | --- |
| **-language** | Programming language (C|C++|C#|Java). **Note:** ddscs v1.1 just generates C++ code. |
| **-ppPath** | C/C++ Preprocessor path. |
| **-ppDisable** | Do not use C/C++ preprocessor. |
| **-replace** | Replace generated files. |

### Output

For the interface declared in the IDL file the following files are created:

|  |  |
| --- | --- |
| **Client** | Client.cxx  <InterfaceName>Proxy.h <InterfaceName>Proxy.cxx  <InterfaceName>ClientRemoteServiceSupport.h  <InterfaceName>ClientRemoteServiceSupport.cxx |
| **Server** | Server.cxx  <InterfaceName>Server.h <InterfaceName>Server.cxx  <InterfaceName>ServerImpl.h <InterfaceName>ServerImpl.cxx  <InterfaceName>ServerRemoteServiceSupport.h  <InterfaceName>ServerRemoteServiceSupport.cxx |
| **Utils** | <InterfaceName>RequestReplyUtils.h  <InterfaceName> RequestReplyUtils.cxx |
| **VS2005 files** | <InterfaceName>-vs2005.sln  <InterfaceName>Server-vs2005.vcproj  <InterfaceName>Client-vs2005.vcproj |
| **Type Support** | <InterfaceName>.h <InterfaceName>.cxx  <InterfaceName>Plugin.h <InterfaceName>Plugin.cxx  <InterfaceName>Support.h <InterfaceName>Support.cxx |
| **Request-Reply**  **Topics Support** | <InterfaceName>RequestReply.h <InterfaceName> RequestReply.cxx  <InterfaceName> RequestReply Plugin.h  <InterfaceName> RequestReply Plugin.cxx  <InterfaceName> RequestReply Support.h  <InterfaceName> RequestReply Support.cxx |

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

Despite of the file quantity generated by the script, as a developer you just should not care about all files.

## Client

Main class in the client side is the proxy, which is named *<InterfaceName>Proxy*. This class is implemented in files *<InterfaceName>Proxy.h* and *<InterfaceName>Proxy.cxx*. The proxy offers to the user the procedures that will call remotely. By example, for the following interface definition in IDL:

interface Bank

{

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

};

User can access to *desposit* procedure in the following way:

BankProxy \*proxy = new BankProxy(0, 4000);

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

DDS\_Long money ;

ReturnCode deposit\_ret ;

DDSCSMessages depositRetValue ;

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

The proxy contains the DomainParticipant entity. This DDS entity can be configured using QoS. The IDL declaration *//#qos* sets the QoS library and QoS profile for the DomainParticipant.

For each procedure inside the interface, it is created a class that inherits from a special class named *ClientRemoteService*. This class contains a DataWriter that sends requests and a DataReader that receives the replies.These DDS entities can be also configured using QoS. The IDL declaration *//#qos-request* sets the QoS library and QoS profile for the DataWriter that sends requests. The IDL declaration *//#qos-reply* sets the QoS library and QoS profile for the DataReader that receives replies. And the IDL declaration *//#qos* sets the QoS library and QoS profile to both DDS entities.

An example of how to invoke the remote procedures using the proxy is generated too. This example is contained in file *Client.cxx*.

## Server

Main class in the server side is the server, which is named *<InterfaceName>Server*. This class is implemented in files *<InterfaceName>Server.h* and *<InterfaceName>Server.cxx*. The server receives all requests and calls to the skeleton. An example of how runs the server is:

BankServer \*server = new BankServer(0, 5);

server->executeServer();

The first parameter is the DDS domain identifier that will use the DomainParticipant. The second parameter is the number of threads that thread’s pool will contain.

The server contains the DomainParticipant entity. This DDS entity can be configured using QoS. The IDL declaration *//#qos* sets the QoS library and QoS profile for the DomainParticipant.

For each procedure inside the interface, it is created a class that inherits from a special class named *ServerRemoteService*. This class contains a DataReader that receives requests and a DataWriter that sends the replies. These DDS entities can be also configured using QoS. The IDL declaration *//#qos-request* sets the QoS library and QoS profile for the DataReader that receives requests. The IDL declaration *//#qos-reply* sets the QoS library and QoS profile for the DataWriterr that sends replies. And the IDL declaration *//#qos* sets the QoS library and QoS profile to both DDS entities. If it is not used any of previous IDL declarations for the procedure, and for the interface it was, then the procedure inherits the interface QoS library and QoS profile.

A skeleton is created as class named *<InterfaceName>ServerImpl*. This class is implemented in files *<InterfaceName>ServerImpl.h* and *<InterfaceName>ServerImpl.cxx*. For each user’s procedure call, this skeleton will be accessed by the server, calling to the requested procedure. The user will implement the procedures behavior in this class. By example, for the following interface definition in IDL:

interface Bank

{

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

};

User must implement the *desposit* procedure inside the following function:

DDSCSMessages

BankImpl::deposit(Account & ac, DDS\_Long money ,ReturnCode &deposit\_ret)

{

DDSCSMessages retCode = OPERATION\_SUCCESSFUL;

return retCode;

}

An example of how to create the server is generated too. This example is contained in file *Server.cxx*.

## Suggested QoS

For the correct behavior of the server and clients, there is a basic QoS profile suggested. This QoS profile sets a reliable communication between the clients and the server. In other case, requests or replies could be lost. It also sets a history that stores all requests and replies.

This QoS profile can be saved in the *USER\_QOS\_PROFILE.xml* file and it can be set using the IDL declaration *//#qos* in the IDL interface.

This QoS profile is:

<qos\_profile name="DefaultQoS\_Profile">

<datawriter\_qos>

<reliability>

<kind>RELIABLE\_RELIABILITY\_QOS</kind>

</reliability>

<history>

<kind>KEEP\_ALL\_HISTORY\_QOS</kind>

</history>

</datawriter\_qos>

<datareader\_qos>

<reliability>

<kind>RELIABLE\_RELIABILITY\_QOS</kind>

</reliability>

<history>

<kind>KEEP\_ALL\_HISTORY\_QOS</kind>

</history>

</datareader\_qos>

</qos\_profile>

# Visual Studio HelloWorld Example

## Write an IDL File.

Write a simple interface named *World* that has a *hello* method:

**World.idl**

*interface Bank*

*{*

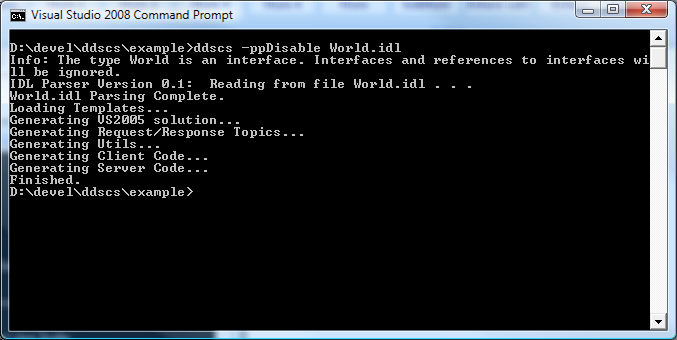
*ReturnCode deposit(in Account ac, in long money);*

*}; //#qos Bank\_Library Bank\_Profile*

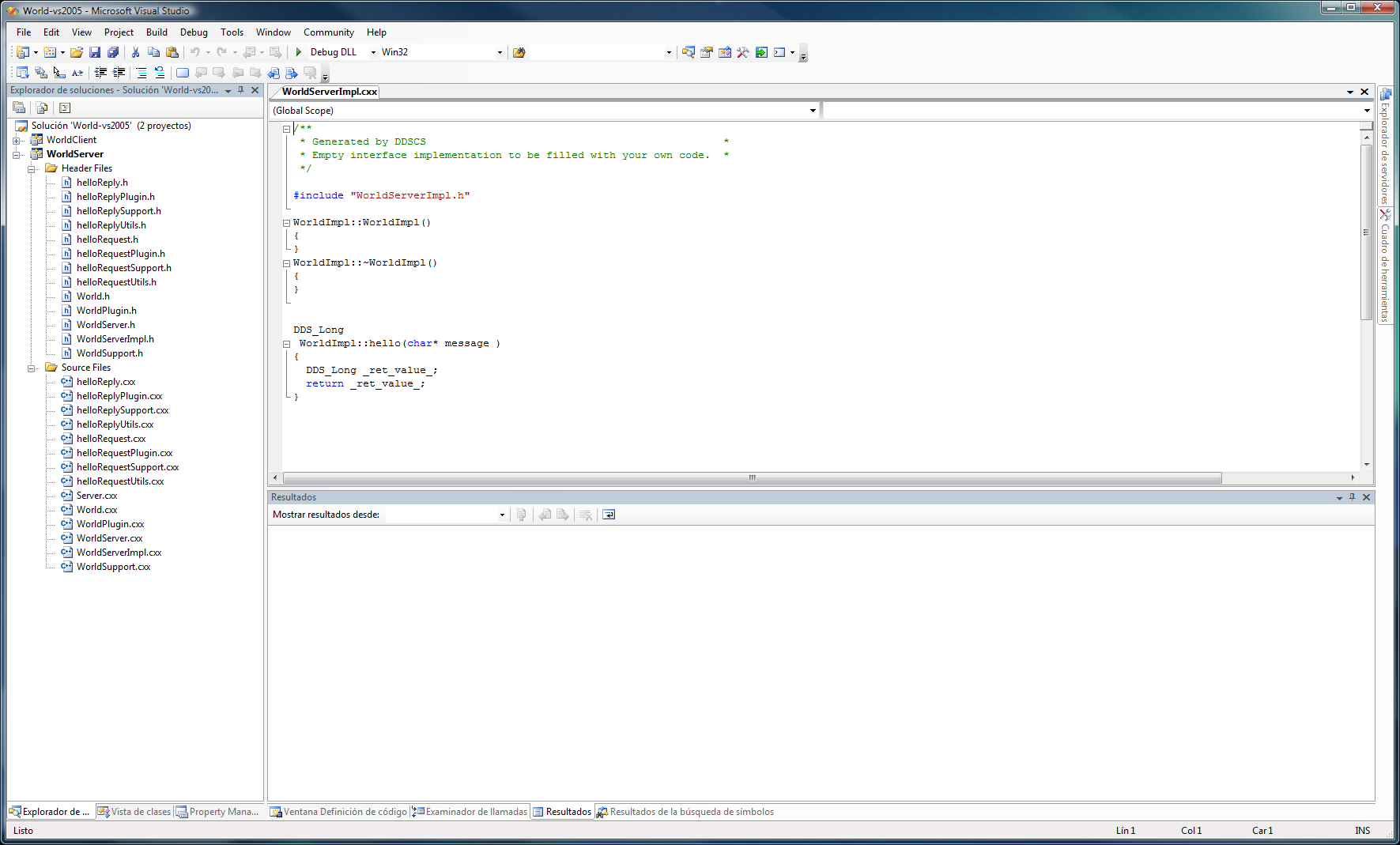
## Execute code generator

Open a command prompt and go to the directory containing World.idl file.

***ddscs -ppDisable World.idl***



## Open VS2005 Solution



## Write your Code

Server Implementation Code

**WorldServerImpl.cxx**

/\*\*

\* Generated by DDSCS

\*

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

\*

\*/

#include "WorldServerImpl.h"

WorldImpl:: WorldImpl()

{

}

WorldImpl::~ WorldImpl()

{

}

DDSCSMessages

WorldImpl::hello(char\* message ,ReturnCode &hello\_ret)

{

DDSCSMessages retCode = OPERATION\_SUCCESSFUL;

hello\_ret = printf("%s\n”, message); // Written by user

return retCode;

}

Client Implementation

**Client.cxx**

/\*\*

\* Generated by DDSCS

\*

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

\*

\*/

#include "WorldProxy.h"

#include "WorldRequestReplyPlugin.h"

int main()

{

int domainId = 0;

unsigned int timeoutInMillis = 4000;

WorldProxy \*proxy = new WorldProxy(domainId, timeoutInMillis);

DDS\_Duration\_t period = {5,0};

char\* message = DDS\_String\_dup("Hello World"); // Writed by user

ReturnCode hello\_ret ;

DDSCSMessages helloRetValue ;

/\*\*

\* Dynamic memory passed to the proxy will be freed before return

\*

\* Pass a copy if you want to keep it

\*

\*/

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

// Written by user

printf("Operation invocation result: %d\n”, helloRetValue);

if(message != NULL) DDS\_String\_free(message);

delete(proxy);

NDDSUtility::sleep(period);

}

The client uses the Proxy pattern to hide all the implementations details. The Proxy is completely thread-safe and many threads can make invocations in parallel.

## Build and execute

Build the solution (F7) and go to <example\_dir>\objs\i86Win32VS2005 directory.

Just double click on WorldServer.exe to start the server and after launch WorldClient.exe. Depending on the chosen configuration to build the example you may need to copy ddscs.dll or ddscsd.dll from %NDDSHOME%\ddscs\lib\ i86Win32VS2005 to the working directory if they are not in the $PATH variable.

