USER MANUAL

DDS Client/Server

Contents

[1. Overview 2](#_Toc252638553)

[1.1. Communication patterns 2](#_Toc252638554)

[1.1.1. Publish/Subscribe 2](#_Toc252638555)

[1.1.2. Client/Server 2](#_Toc252638556)

[1.1.3. Peer to Peer 2](#_Toc252638557)

[1.2. Middleware Selection 2](#_Toc252638558)

[1.3. Client/Server Communications with DDS 3](#_Toc252638559)

[1.3.1. Generic Remote Procedure Call with DDS. 3](#_Toc252638560)

[2. Installation 4](#_Toc252638561)

[2.1. Installing RTI DDS 4](#_Toc252638562)

[2.2. Installing DDS Client/Server Extension 9](#_Toc252638563)

[2.3. DDSCS Runtime Libraries 9](#_Toc252638564)

[2.4. Microsoft Runtime Libraries 9](#_Toc252638565)

[3. Usage 10](#_Toc252638566)

[4. DDSCS Output 10](#_Toc252638567)

[4.1. Client Side 11](#_Toc252638568)

[4.2. Server Side 11](#_Toc252638569)

[5. Visual Studio HelloWorld Example 11](#_Toc252638570)

[5.1. Write an IDL File. 11](#_Toc252638571)

[5.2. Execute ddscs 11](#_Toc252638572)

[5.3. Open VS2005 Solution 12](#_Toc252638573)

[5.4. Write your Code 12](#_Toc252638574)

[5.5. Build and execute 13](#_Toc252638575)

[6. Limitations 14](#_Toc252638576)

# 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.



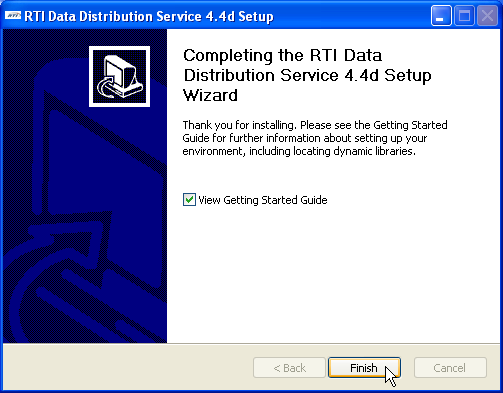
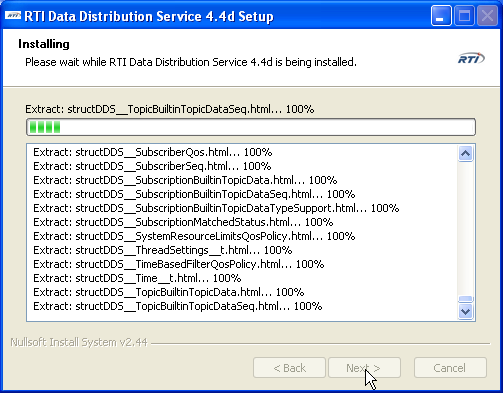
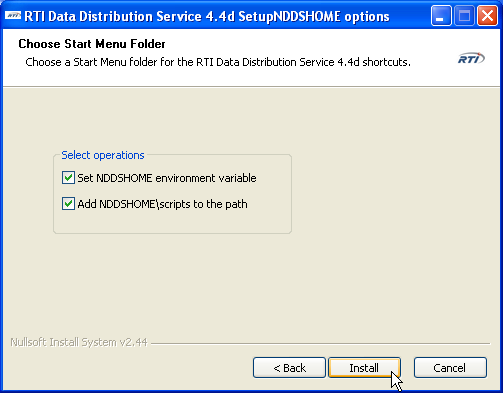
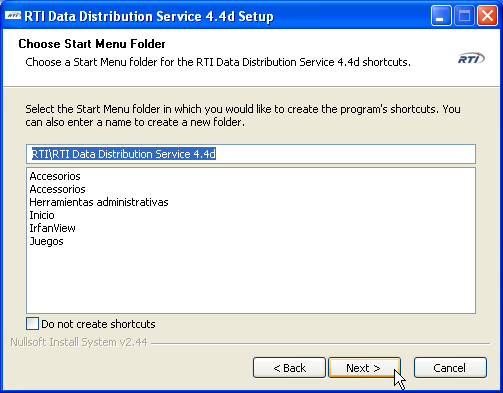
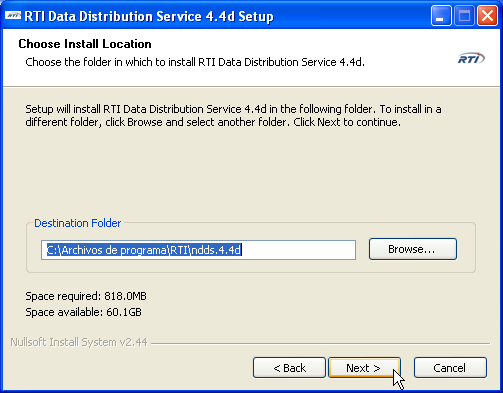
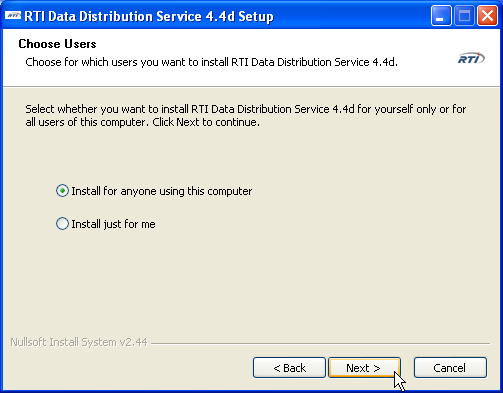
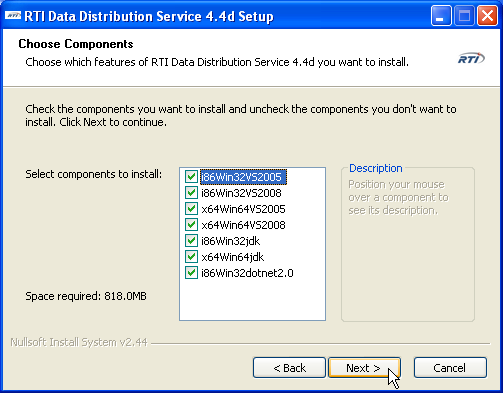
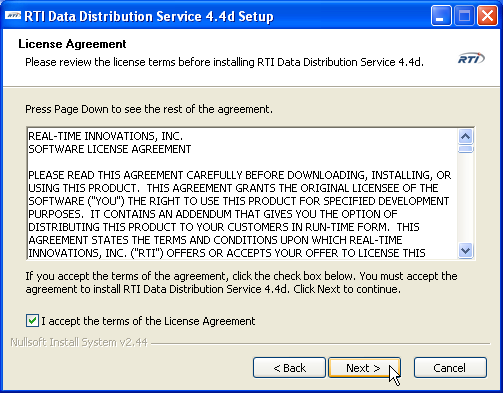
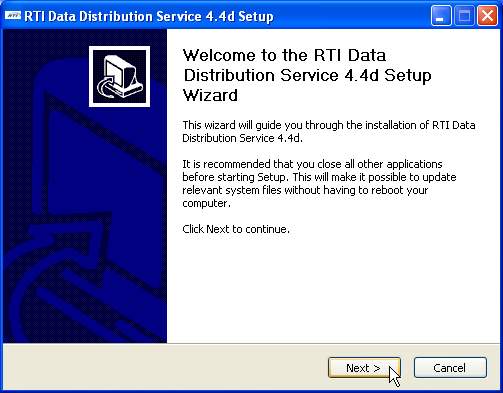
Figure 1: Generic Remote Procedure Invocation DDS style.

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

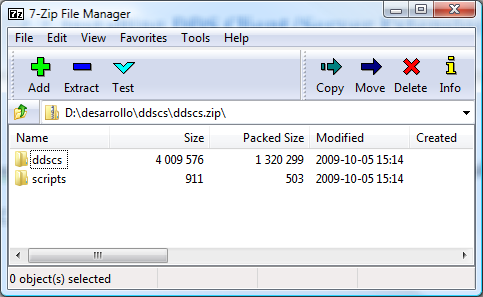
## Installing RTI DDS

First step before using the Client/Server plugin is the installation of the RTI DDS 4.4d middleware. Execute the installer *rtidds44d-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*.zip*



and extract it over the RTI DDS installation folder:

*C:/Archivos de programa/RTI/ndds.4.4d*

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 28-07-2009 and, consequently, the Visual Studio Redistributable package of 28-07-2009 is required. It can be found in:

Visual 2005:

<http://www.microsoft.com/downloads/details.aspx?displaylang=es&FamilyID=766a6af7-ec73-40ff-b072-9112bab119c2>

Visual 2008:

<http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=2051a0c1-c9b5-4b0a-a8f5-770a549fd78c>

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

# Usage

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

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

This application accepts some arguments.

|  |  |
| --- | --- |
| **-language** | Programming language (C|C++|C#|Java). **Note:** ddscs v1.0 just generates C++ code. |
| **-ppPath** | C/C++ Preprocessor path. |
| **ppDisable** | Do not use C/C++ preprocessor. |
| **-replace** | Replace generated files. |

# DDSCS Output

For the interface declared (see limitations) 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 |

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

Despite of the file quantity, as a developer you just should care about the following ones.

## Client Side

* Client.cxx : Just an example of how to invoke a remote procedure using the proxy. Just replace it with you own application main.
* <InterfaceName>Proxy: A local object with the same API as the server. Just call the local functions and the proxy will take care of all.

## Server Side

* Server.cxx : Just an example of how to start the server. Replace it with your own main or use it as is.
* <InterfaceName>ServerImpl: Server implementation. Put here your server logic, taking account that the methods of this class may be called by several threads at the same time.

# Visual Studio HelloWorld Example

## Write an IDL File.

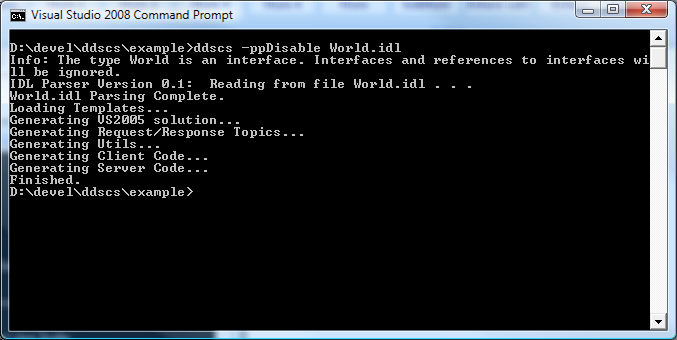
Write a simple interface named *World* that has a *hello* method:

|  |
| --- |
| World.idl  interface World  {  long hello(in string message);  }; |

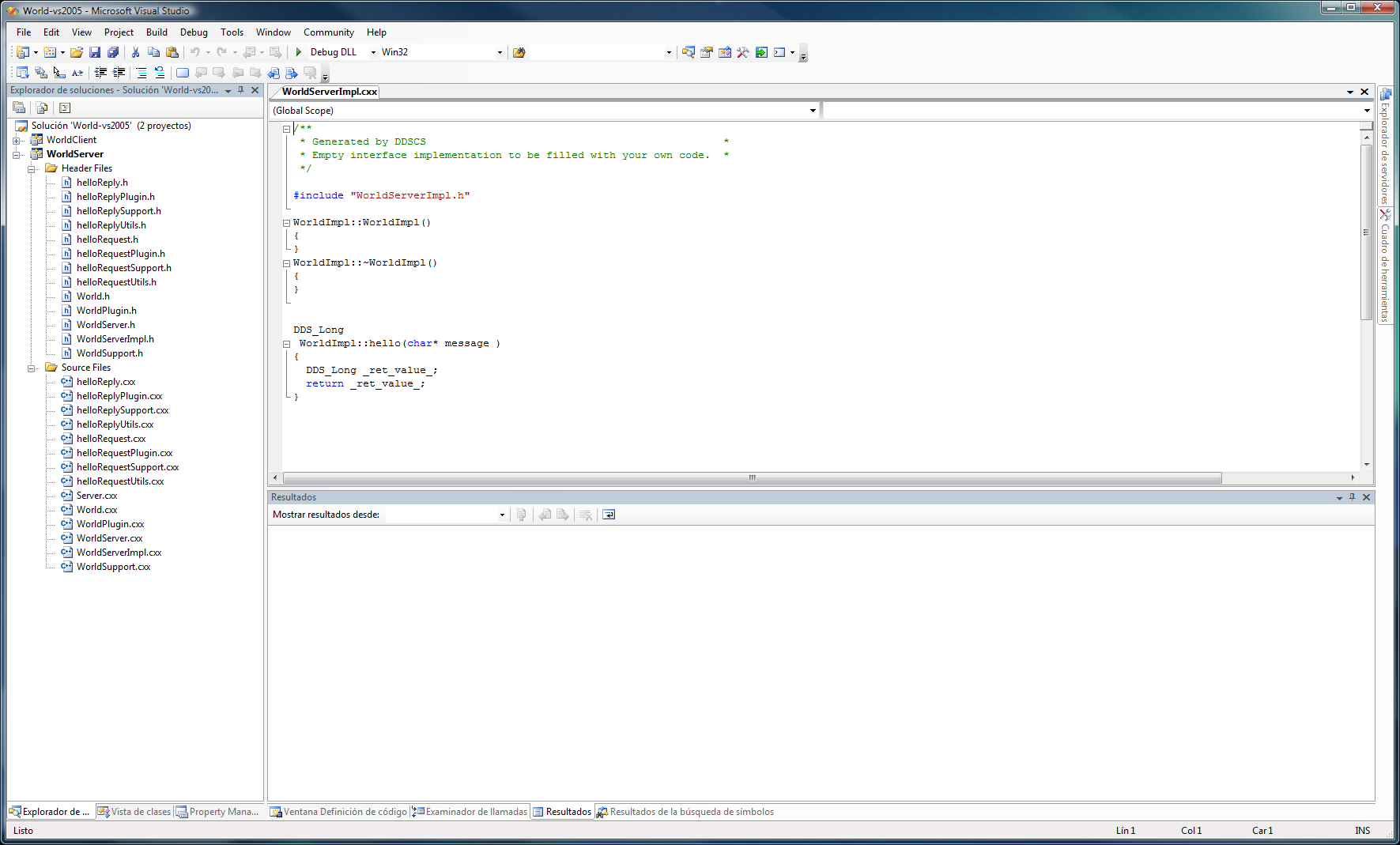
## Execute ddscs

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()  {  }    DDS\_Long WorldImpl::hello(char\* message )  {  DDS\_Long \_ret\_value\_ = printf("%s\n", message);  return \_ret\_value\_;  } |

Client Implementation

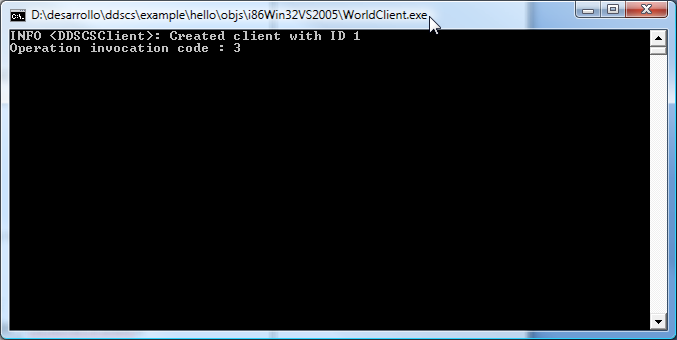
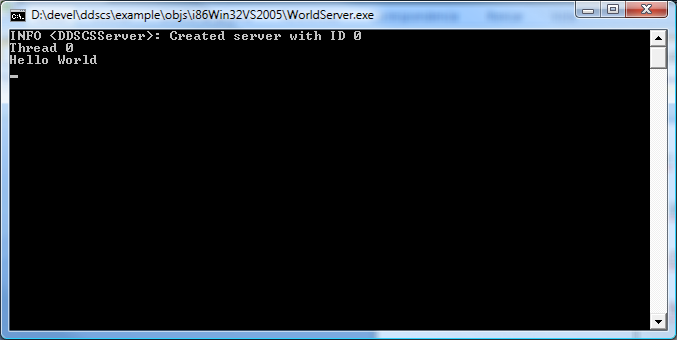
|  |
| --- |
| Client.cxx  /\*\*  \* Generated by DDSCS \*  \* Example client. Method params should be initialized before execution \*  \*/  #include "WorldProxy.h"  #include "WorldPlugin.h"  int main()  {  int domainId = 0;  unsigned int timeoutInMillis = 4000;  WorldProxy \*proxy = new WorldProxy(domainId, timeoutInMillis);  DDS\_Duration\_t period = {5,0};    char\* message = DDS\_String\_dup("Hello World");  DDS\_Long 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);  printf("Operation invocation result : %d\n", helloRetValue);  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.



# Limitations

DDS Client/Server Extension has, as of version 1.0, some limitations:

* It can handle just one interface per IDL file.
* The interface and the IDL MUST have the same name.
* Complex types (sequences, bounded strings, unions…) must be previously named using *typedef* keyword, as IDL 3.0 specification enforces.
* Simple types (long, octet…) must NOT be redefined using *typedef* keyword.
* No namespace (*module* keyword) support yet.
* Dynamic memory passed to the proxy will be freed before return, which is not exactly the expected behavior for input params. A copy should be passed.