DDS CLIENT/SERVER DETAILED DESIGN

Contenido

[1 Introduction 2](#_Toc327980402)

[1.1. Target 2](#_Toc327980403)

[1.1 Audience 2](#_Toc327980404)

[1.2 Related documentation 2](#_Toc327980405)

[1.3 Definitions and acronyms 2](#_Toc327980406)

[2 Static structure model 2](#_Toc327980407)

[2.1 External tool 2](#_Toc327980408)

[2.1.1 DDS types 2](#_Toc327980409)

[2.1.2 Client side 5](#_Toc327980410)

[2.1.3 Server side 6](#_Toc327980411)

[2.2 Client 7](#_Toc327980412)

[2.2.1 Threading issues 9](#_Toc327980413)

[2.3 Server 10](#_Toc327980414)

[3 Behavior model 12](#_Toc327980415)

[3.1 Client 12](#_Toc327980416)

[3.1.1 Client initialization and calling a remote function 12](#_Toc327980417)

[3.2 Server 13](#_Toc327980418)

[3.2.1 Server initialization and receiving a client's request 13](#_Toc327980419)

[4 Unit test design 14](#_Toc327980420)

[5 External Tool design 14](#_Toc327980421)

[5.1 IDL Parser 15](#_Toc327980422)

[5.2 Code Generation 15](#_Toc327980423)

[5.3 Types Generation 15](#_Toc327980424)

# Introduction

## Target

The purpose of this document is the design of a client/server architecture over the RTI DDS middleware. This project tries to give the possibility of use the DDS middleware like a client/server application. A user is able to create client applications which can execute functions of a remote server, all this over DDS middleware at low level. For more information read the document 1 in related documentation section.

## Audience

This document has been written for the developers of this project.

## Related documentation

1. Avanza proposal about this project (eProsima\_DDS-CS\_Avanza\_v1.doc)

## Definitions and acronyms

**DDS:** **Data Distribution Service** for Real-time Systems is a specification of a [publish/subscribe](http://en.wikipedia.org/wiki/Publish/subscribe) [middleware](http://en.wikipedia.org/wiki/Middleware) for [distributed systems](http://en.wikipedia.org/wiki/Distributed_system) created in response to the need to standardize a [data-centric](http://en.wikipedia.org/wiki/Database-centric_architecture) publish-subscribe programming model for distributed systems.

# Static structure model

The proposed model in this document tries to give an easy creation of a client/server application for the user. The internal use of subscriber/publisher architecture must be transparent. The user only wants to execute remote functions and obtain the results. The user should only develop the remote functions that are executed in the server, and the code that calls these functions in the client part.

Then this project is divided in three components: the client, the server and a tool that generates specific code that creates the message communication in DDS data types.

## External tool

The external tool reads an IDL file and generates C++ code. In this IDL must be defined an interface and the remote procedures. The external tool generates all necessary code, including client’s side and server’s side.

### DDS types

As this project uses the DDS middleware, the last information that is sent and received is DDS types. Then external tool generates a new IDL with new DDS Types. This new IDL is processed by the *rtiddsgen* application.

This external tool transforms the input parameters of each procedure in a new DDS type. By example, for the next procedure:

interface Example1

{  
 long function1(in short param1, in long param2, out long param3);  
};

External tool generates next DDS Types:

struct function1tRequest{

unsigned long clientServiceId[4]; //@key

unsigned long numSec;

short param1;

long param2;

}; //@top-level true

Also the tool transforms the output parameters of each function in a new DDS type. Using the previous example, external tool generates next DDS Types:

struct function1Reply{

unsigned long serverServiceId[4]; //@key

unsigned long clientServiceId[4]; //@key

unsigned long numSec;

long ddscsRetCode;

long param3;

long returnedValue;

}; //@top-level true

*rtiddsgen* application generates the code of new DDS types. Using the previous example, this code is stored in next file: *Example1RequestReply.cxx*. Besides creating DDS types, the tool generates the code that serializes/deserializes the DDS type. In the example this code is in next files: *Example1RequestReplyPlugin.cxx* and *Example1RequestReplySupport.cxx*.

Furthermore, the external tool generates the code that fills the DDS type data with the parameter's values of the user's function. In the example this code is in *Example1RequestReplyUtils.cxx* file. For input parameters of each procedure, this file contains a class that has a function to register the DDS type, a function to create a new instance of the DDS type using the user data and a function to extract the user data from the DDS type. Using the previous example, the code generated is:

const char\*   
function1RequestUtils::registerType(DDSDomainParticipant \*clientParticipant)

{

const char \*typeName = NULL;

if(clientParticipant != NULL)

{

typeName = function1RequestTypeSupport::get\_type\_name();

if(getOctetRequestTypeSupport::register\_type(  
 clientParticipant, typeName) != DDS\_RETCODE\_OK)

{

return NULL;

}

}

return typeName;

}

function1Request\* function1RequestUtils::createTypeData(DDS\_Short param1, DDS\_Long param2)

{

function1Request\* instance =  
 function1RequestTypeSupport::create\_data();

instance->param1 = param1;

instance->param2 = param2;

return instance;

}

void function1RequestUtils::extractTypeData(function1Request\* data , DDS\_Short &param1, DDS\_Long & param 2)

{

param1 = data->param1;

param2 = data->param2;

}

For output parameters of each procedure, this file contains class that has a function to register the DDS type, a function to create a new instance of the DDS type using the user data and a function to extract the user data from the DDS type. Using the previous example, the code generated is:

const char\* function1ReplyUtils::registerType(DDSDomainParticipant \*clientParticipant)

{

const char \*typeName = NULL;

if(clientParticipant != NULL)

{

typeName = function1ReplyTypeSupport::get\_type\_name();

if(function1ReplyTypeSupport::register\_type(   
 clientParticipant, typeName) != DDS\_RETCODE\_OK)

{

return NULL;

}

}

return typeName;

}

function1Reply\* function1ReplyUtils::createTypeData(DDS\_Long param3, DDS\_Long& returnedValue)

{

Function1Reply\* instance =   
 funcion1ReplyTypeSupport::create\_data();

instance->param3 = param3;

instance->returnedValue = returnedValue;

return instance;

}

void function1ReplyUtils::extractTypeData(function1Reply\* data , DDS\_Long &param3, DDS\_Long& returnedValue)

{

param3 = data->param3;

returnedValue = data->returnedValue;

}

### Client side

The external tool also generates the code for the client side. For each remote procedure defined in the IDL, it is generated a class that contains specific code. This class has a method to write a request and a method to take the reply using DDS entities. In this document, these classes are known as the remote service of the procedure.   
The external tool doesn’t generate the creation of DDS entities. This is encapsulated in the DDS Client/Service library as it will be seen later, by a class that the remote service inherits. Using the previous example, this class has this form:



The external tool generates a client proxy that manages the previous remote services (one for each procedure). This proxy is used by the user to call the remote procedure. The class contains the code that transforms the procedure’s parameters to DDS types and uses the remote service of the procedure to send the DDS type and receive the reply. This proxy’s class has this form:



### Server side

The external tool also generates the code for the server side. For each remote procedure defined in the IDL, it is generated a class that contains specific code. This class has a method to take a request and a method to write using DDS entities. In this document, these classes are known as the remote service of the procedure.  
The external tool doesn’t generate the creation of DDS entities. This is encapsulated in the DDS Client/Service library as it will be seen later, by a class that the remote service inherits. Using the previous example, this class has this form:



The external tool generates a server core that manages the previous remote services (one for each procedure). This core is responsible of calling to the skeleton that user will implement. The class uses the remote service of the procedure to receive the DDS types and send the reply, and it contains the code that transforms the DDS types to procedure’s parameters before calling the skeleton. This proxy’s class has this form:



Finally, the external tool generates the skeleton that server calls when receives a request. The user will have to implement the functions of this skeleton. The skeleton has this form:



## Client

The client's objective is to send the user's request and receive the server's reply. The client’s frontend for the user is the client’s proxy. The client’s proxy offers the remote procedures to the user. The user will call a client's proxy procedure with its parameters. Since the user calls the procedure, the user hasn't to know what's going on. Then, in the class model described below, some classes are generic and are developing one time. These classes are in the product's library. Other classes are templates that the external tool creates depending on the functions defined by the user.



The *function1RequestUtils* and *function1ReplyUtils* are classes that are generated by the external tool. *function1RequestUtils* is generated from the DDS type of the input parameters of the remote procedure. *function1ReplyUtils* is generated from the DDS type of the output parameters of the remote procedure. These classes know how to create a DDS type data with the parameter's values of the remote procedure. This functionality is implemented in the function *createTypeData().* Also these classes know how to extract data from DDS type data to fill he output parameters. This functionality is implemented in the function *extractTypeData()*.

The *Exampl1Proxy* class is generated with the external tool. It contains the remote procedures that the user can use. These functions are defined in the language which is used by the user for coding his application. The user only have to call these functions. When a function is called, the *Example1Proxy* class uses the \**RequestUtils* class to create a filled type data with the input parameters using the function *createTypeData()*. Then uses the function *execute ()* to send the request type data. This function returns the reply type data and the *Example1Proxy* class uses the \**ReplyUtils* class to return the output parameters using the function *extractTypeData()*.

For each remote procudure there is a *ClientRemoteService* class (inherited by a class generated by the external tool, in the example is *function1ClientRemoteService*) . The *ClientRemoteService* objects are created in the client's proxy initialization. Each object manages the DDS objects that each remote function needs to communicate with the server. It manages the publisher of the request data, the data writer of the request data, the subscriber of the reply data and the data reader of the reply data. Furthermore, this class register the request type and reply type at the initialization, and it creates both topics. The *ClientRemoteService* object receives from the client’s proxy the request data that it has to send. The *ClientRemoteService* awaits the server reply and returns the status. This functionality is implemented by the function *execute()*. The *ClientRemoteService* can obtain the server reply with the function *takeReply().*

### Threading issues

When a user calls a remote procedure, there is only one global variable that is used. The rest of variables and objects are created in the call, except DDS entities that are thread safe. That global variable is *m\_numSec* that store the next sequence number that request should use. This variable is protected by a mutex.

## Server

The server's objective is to receive a client's request and send a reply after the called function's execution. The user only needs to know how his functions are done.



The *function1RequestUtils* and *function1ReplyUtils* are classes that are generated by the external tool. *function1RequestUtils* is generated from the DDS type of the input parameters of the remote procedure. *function1ReplyUtils* is generated from the DDS type of the output parameters of the remote procedure. These classes know how to create a DDS type data with the parameter's values of the remote procedure. This functionality is implemented in the function *createTypeData().* Also these classes know to extract data from DDS type data to fill he output parameters. This functionality is implemented in the function *extractTypeData()*.

The *Example1Server* class inherits from the *DDSCSServer* class. This class is generated by the external tool and implements a specific function for each remote procedure. In the example the function will be *function1()*. When *Example1Server* receives a client request of a remote procedure, this function is called by *Example1Server* and it uses the *function1RequestUtils* to extract the parameters of the remote procedure. Then it calls the user’s implementation of the remote procedure and transforms the output parameters in a reply data instance using *function1ReplyUtils* class.

For each remote procedure there is a *ServerRemoteService* class (inherited by a class generated by the external tool, in the example is *function1ServerRemoteService*). The *ServerRemoteService* objects are created in the server's initialization. Each object manages the DDS objects that each remote function needs to communicate with the clients. It manages the publisher of the reply data, the data writer of the reply data, the subscriber of the request data and the data reader of the request data. Furthermore, this class register the request type and reply type at the initialization, and it creates both topics. The *ServerRemoteService* awaits the client request and notify to the *Example1Server (DDSCSServer)* using the function *schedule()*. The *ServerRemoteService* object receives from the *Example1Server* object the reply data that it has to send. This functionality is implemented by the function *sendReply()*.

The external tool generates the skeleton that user has to implement. In the example this class is *Example1ServerImpl*. This skeleton is generated with one function for each remote procedure. These functions are the last function, and it has to have the implementation of the remote procedure. These functions are called by it equals in the *Example1Server* class.

# Behavior model

In this section is shown some functionalities from sequence diagrams.

## Client

### Client initialization and calling a remote function



## Server

### Server initialization and receiving a client's request



# Unit test design

# External Tool design

First of all, the external tool (idl2ddscs?) has to parser the idl file which contains the service.

For each method of each interface the outputs shown in the figure below would be generated.



The tool, then will use the DDS Framework utils to generate Request and Anwser dds types and (de)marshalling helpers.

## IDL Parser

Two alternatives have been considered for the parser:

* JavaCC and JJTree
* Antlrv3

JavaCC seems to use simpler grammar and the generated parser are more readable and easier to modify if required.

Antlrv3 seems to be harder to learn but also should be more powerful, providing tree walkers for tree transformations and grammar integrated with StringTemplates, the chosen tool for code and IDL generation.

Finally, we have decided to use JavaCC for its less time to market and also because it would be easier for other people to maintain the project if necessary.

The parser will generate a tree with only the information required for generation.

One interesting option of JavaCC is that it can generate a tree whose nodes implement the visitor pattern, making easier to write different visitors that will traverse the tree generating different things from the same information.

## Code Generation

StringTemplates is the tool for generating the C++ Code for Server and Client sides.

For each interface found in the idl specification a client proxy class (InterfaceNameProxy) a server skeleton (InterfaceNameSkeleton) and an Implementation class (InterfaceNameServerImpl) which has only empty functions to be filled with the server behaviour are generated.

The tool, in its current version, will ignore inherited interfaces.

## Types Generation

For each method two idl files have to be generated, also using StringTemplates tool:

* MethodNameRequestType.idl: a struct whose members are all in and inout parameters of the method.
* MethodNameAnswerType.idl: a struct whose members are all out and inout parameters of the method.

If any parameter depends on other user defined types, those types will be also included. User defined types handling optimizations will be analyzed later on, if possible.

The idl files will then be processed by DDS Framework generation tool (ddsgen) .