DDS CLIENT/SERVER DETAILED DESIGN

Contenido

[1. Introduction 2](#_Toc239762427)

[1.1. Target 2](#_Toc239762428)

[1.2. Audience 2](#_Toc239762429)

[1.3. Related documentation 2](#_Toc239762430)

[1.4. Definitions and acronyms 2](#_Toc239762431)

[2. Static structure model 2](#_Toc239762432)

[2.1. External tool 2](#_Toc239762433)

[2.2. Client 3](#_Toc239762434)

[2.3. Server 5](#_Toc239762435)

[3. Behavior model 6](#_Toc239762436)

[3.1. Client 7](#_Toc239762437)

[3.1.1. Client initialization and calling a remote function 7](#_Toc239762438)

[3.2. Server 8](#_Toc239762439)

[3.2.1. Server initialization and receiving a client's request 8](#_Toc239762440)

[4. Unit test design 9](#_Toc239762441)

[5. External Tool design 9](#_Toc239762442)

[5.1. IDL Parser 9](#_Toc239762443)

[5.2. Code Generation 10](#_Toc239762444)

[5.3. Types Generation 10](#_Toc239762445)

# 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. The internal use of subscriber/publisher architecture must be transparent. The user only want 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 IDL files where are defined the remote functions. As this project uses the DDS middleware, the last information that is sent and received is DDS types. This external tool transforms the input parameters of each function in a new DDS type, by example: uFunction1RequestType. Also the tool transforms the output parameters of each function in a new DDS type, by example: uFunction1ReplyType.

//TODO Definicion de los tipos creados.

Then the tool generates the code of the DDS types. In the example this code is in *uFunction1RequestType.cxx* and *uFunction1ReplyType.cxx* files. Besides creating DDS types, the tool generates the code that serializes/deserializes the DDS type. In the example this code is in *uFunction1RequestTypePlugin.cxx*, *uFunction1RequestTypeSupport.cxx*, *uFunction1ReplyTypePlugin.cxx* and *uFunction1ReplyTypeSupport.cxx* files. Furthermore, the 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 *uFunction1RequestTypeUtils* and *uFunction1ReplyTypeUtils* classes.

## Client

The client's objective is to send the user's request and receive the server's reply. The user calls a client's function with certain parameters. Since the user calls the function, 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 *uFunction1RequestTypeUtils* and *uFunction1ReplyTypeUtils* are classes that are generated by the external tool. *uFunction1RequestTypeUtils* is generated from the DDS type of the input parameters of the user's function. *uFunction1ReplyTypeUtils* is generated from the DDS type of the output parameters of the user's function. These classes know how to create a DDS type data with the parameter's values of the user's function. 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 *Proxy* class is generated with the external tool. It contains the remote functions 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 *Proxy* class uses the *RequestTypeUtils* to create a filled type data with the input parameters using the function *createTypeData()*. Then uses the function *executeRemoteService()* to send the request type data. This function returns the reply type data and the *Proxy* class uses the ReplyTypeUtils to return the output parameters using the function *extractTypeData()*.

For each user's functions there is a *ClientRemoteService* class. The *ClientRemoteService* objects are created in the client's 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* object 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 *Client*  can obtain the server reply with the function *getReply().*

The *Client* class manages the *ClientRemoteService* objects. When a request data has to be sent, the *Client* object searches the *ClientRemoteService* associated with the request data.

### Threading issues

Each *ClientRemoteService* object have internally a *DataWriter* for the Request Topic and a *DataReader*  for the Reply topic. It uses a content filter on the reply topic based on the clientId and the sequenceNumber to select the correct answer. So, the execute() and getReply() methods must be called atomically , preventing for multiple threads make requests using the same *ClientRemoteService* object. The generated code enforces this.

Please, note that a *Proxy* object can have several *ClientRemoteService* objects, one for each method defined on the IDL interface, and different threads can make requests through different *ClientRemoteService* objects at the same time.

#### Threading improvements.

For next releases *ClientRemoteService* will be made thread safe. The problem that prevents it actually is that this class uses a trick that lets it read individual samples from any type (no *FooDataReader*), but can’t read sequences of samples. So it just can use the read\_next\_sample or take\_next\_sample methods.

Three main design options are under study for achieving this.

* Generating code: if the *ClientRemoteService* is partially generated, it can use specific FooDataReaders and iterate or use DDSQueryConditions for selecting the reply to each thread Request.
* Creating a custom DataReader class: This class would inherit from DDSDataReader class and would have access to the protected read and take untyped methods. It may also use DDSQueryConditions.
* Using Dynamic Data API: This option was initially rejected because it has major impact onto the volume of generated code, and also most of the development was previous to ndds 4.4 release. It surely deserves a second chance.

## 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 *uFunction1RequestTypeUtils* and *uFunction1ReplyTypeUtils* are classes that are generated by the external tool. *uFunction1RequestTypeUtils* is generated from the DDS type of the input parameters of the user's function. *uFunction1ReplyTypeUtils* is generated from the DDS type of the output parameters of the user's function. These classes know how to create a DDS type data with the parameter's values of the user's function. 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 *ServerTemplate* class inherit from the *Server* class. This class is generated by the external tool and implements the function *executeFunction()*. This function receives a request data instance and uses the *uFunction1RequestTypeUtils* to extract the parameters of the user's function. Then it calls the user's function and transforms the output parameters in a reply data instance using *uFunction1ReplyUtils* class.

For each user's functions there is a *ServerRemoteService* class. 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 server. 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 *Server*. The *ServerRemoteService* object receives from the *Server* object the reply data that it has to send. This functionality is implemented by the function *executeFunction()*. The *Server*  can obtain the client request with the function *getRequest().*

### Threading issues

The *Server* class manages the *ServerRemoteService* objects. When a request data has been received, the *Server* object is notified by the *ServerRemoteService*  associated with the request data. Then the server assign the new task to a thread using the *ThreadManager.*

All threads, which have to be created for replies the client's requests, are managed by the *ThreadManager* class. The *ThreadManager* implementes a thread pool.

When a new task is assigned to a *Thread*, it calls the function *receiverRequest()*.

# 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

We assume that the DDS middleware is free of bugs. In the unit test design, mocks are created for the DDS middleware object and structures. The generic model of these mocks are shown in the next image.



Before a unit test execution, a value to be returned can be given for each function in the mock. After the execution the number of calls for each function in the mock can be requested.

With the mocks we can check the external relation with the DDS middleware for each tested class.

# 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) .