RPC4DDS

DETAILED DESIGN

Content

[1 Introduction 2](#_Toc329782669)

[1.1. Target 2](#_Toc329782670)

[1.1 Audience 2](#_Toc329782671)

[1.2 Related documentation 2](#_Toc329782672)

[1.3 Definitions and acronyms 2](#_Toc329782673)

[2 Static structure model 2](#_Toc329782674)

[2.1 IDL compiler 3](#_Toc329782675)

[2.1.1 DDS types 3](#_Toc329782676)

[2.1.2 Client side 6](#_Toc329782677)

[2.1.3 Server side 7](#_Toc329782678)

[2.1.4 Project files 8](#_Toc329782679)

[2.2 Client 9](#_Toc329782680)

[2.2.1 Threading issues 10](#_Toc329782681)

[2.3 Server 11](#_Toc329782682)

[2.4 Portability 12](#_Toc329782683)

[3 Behavior model 12](#_Toc329782684)

[3.1 Client 13](#_Toc329782685)

[3.1.1 Client initialization and calling a remote function 13](#_Toc329782686)

[3.2 Server 15](#_Toc329782687)

[3.2.1 Server initialization and receiving a client's request 15](#_Toc329782688)

[4 Unit test design 16](#_Toc329782689)

[5 External Tool design 16](#_Toc329782690)

[5.1 IDL Parser 17](#_Toc329782691)

[5.2 Code Generation 17](#_Toc329782692)

[5.3 Types Generation 17](#_Toc329782693)

# Introduction

## Target

The purpose of this document is the design of RPC framework that uses the standard DDS for communications. This project tries to give the possibility of use a DDS middleware as a client/server application. A user is able to create client applications which can execute services exposed in a remote server, all this over a DDS middleware at low level. Also a user is able to create server applications which expose services that can be accessed by remote clients.

## 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)
2. Requirement documentation (Requirements.docx)

## 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 RPC framework for the user. The internal use of subscriber/publisher architecture must be transparent. The user only wants to execute remote services and obtain the results.

As the standard DDS is used, the concept of remote procedure call is mapped in sending a request from client and receiving a reply from server. The data used to construct the request and the reply should be created using DDS types. This RPC framework offers two APIs: a client-side API and a server-side API. With the former, the user is able to send a request (as making a remote procedure call) and receive the reply (as obtaining the results of the remote procedure call). With the latter, the user is able to receive a request and send the reply.

As any RPC framework, user should only develop the implementation of remote services that are exposed in the server and the code that calls these services in the client side. For this purpose, a user shall be able to define the remote services in an easy way. Interface Definition Language is supported by this RPC framework to define remote services. A tool is responsible for reading the IDL and generating the request and reply types (as DDS types) and the specific source code to use the client-side API and the server-side API.

This project is divided in three components:

* Client API: This API offers a mechanism to call remote services.
* Server API: This API offers a mechanism to implement remote services.
* IDL compiler: This tool reads a IDL file, where is defined the remote services, and generates the DDS data types and specific source code.

This project is oriented to use standard DDS API and standard DDS types. The main supported DDS implementation is RTI DDS and the examples in this document are based in its naming. OpenDDS will be supported by this project too.

Note: In a future this design shall support several replies from one request.

## IDL compiler

This tool reads an IDL file and generates specific source code. This source code can be generated in different programming languages. The main supported programming language is C++, but it will be also supported Java and .NET.

IDL compiler is implemented in Java, being more easily run on different operating systems. For the purpose of supporting CORBA 2.x IDL, the IDL compiler is designed using a SUN IDL grammar (located in file *idl.jjt*). This grammar is transformed to Java code using *jjtree*. The implementation of the generated grammar supports the next IDL characteristics:

* Basic types: octet, char, wchar, short, unsigned short, long, unsigned long, long long, unsigned long long, float, double, boolean, enumerations, strings and wide-strings.
* Complex types: arrays, sequences, type definitions, unions and structures.
* One interface with several operations.
* *oneway* term is supported.

An interface and the remote procedure must be defined in the IDL file. The IDL compiler generates all necessary code, including client’s side and server’s side. IDL compiler is implemented using Java StringTemplates to minimize the code.

### DDS types

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

IDL compiler 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; //@key

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; //@key

long ddsrpcRetCode;

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 IDL compiler 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 IDL compiler 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 IDL compiler doesn’t generate the creation of DDS entities. This is encapsulated in the RPC4DDS 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 IDL compiler 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 offers to the user the remote procedure as a function member of the proxy and this proxy’s class has this form:



For each remote procedure IDL compiler also generates a function member for the proxy that has a asynchronous behavior.

### Server side

The IDL compiler 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 IDL compiler doesn’t generate the creation of DDS entities. This is encapsulated in the RPC4DDS 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 IDL compiler 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 server’s class has this form:



Finally, the IDL compiler 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:



### Project files

IDL compiler not only generates the specific source code, but it generates the project files necessary for compiling the generated source code. Depending on the operation system selected by the user, the project files can change.

* In Windows, IDL compiler generates project file for Visual Studio 2010.
* In Linux, IDL compiler generates Makefiles.

## 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 IDL compiler creates depending on the functions defined by the user.



The *function1RequestUtils* and *function1ReplyUtils* are classes that are generated by the IDL compiler. *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()*.

As RPC4DDS supports two type of transport (TCP or UDP), the IDL compiler generates two possible frontends for the user. *Example1WANProxy* uses a *TCPTransport* object that allows a TCP connection with a server. *Example1Proxy* uses a *UDPTransport* object that uses the UDP discovery of DDS. Both proxies inherits from the class *Example1ProxyH*. The *Example1ProxyH* class is also generated with the IDL compiler. 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()*.

The *Example1ProxH* class derived from *Client* class. When this class is created, a object of class *AsyncThread* is created too. This object creates and manages a separated thread and a DDS WaitSet used in asynchronous invocations. IDL compiler generates an extra function for each remote procedure. This function has the same name as synchronous invocation but with *\_async­* prefix. This function will create a *function1Task* object, which class is also generated by IDL compiler, and adds the task to *AsyncThread*.

For each remote procedure there is a *ClientRemoteService* class (inherited by a class generated by the IDL compiler, 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. *ClientRemoteService* supports several types of invocation:

* Synchronous invocation: The *ClientRemoteService* sends the request, it awaits the server reply and returns the status. *ClientRemoteService* creates a DDS WaitSet and a DDS QueryCondition to wait the server reply. This functionality is implemented by the function *execute()*. The *ClientRemoteService* can obtain the server reply with the function *takeReply()*.
* Asynchronous invocation: The *ClientRemoteService* sends the request but doesn’t await the server reply. This invocation is non-blocking for the user’s thread. Each remote procedure’s function, generated by the IDL compiler, will have a similar function with the prefix *\_async*. This new function accepts a callback that will be called when request is received. The object *AsyncThread* is responsible for manages the tasks generated by asynchronous invocations and it receives the replies from servers.
* One-way invocation: The *ClientRemoteService* sends the request but doesn’t await the server reply. This kind of invocation has to set by user in the IDL file, using the special word *oneway*. Operation definitions with this word cannot have any out parameter.

### API

User has to create an object of the generated proxy to be able to use the defined interface by him. The constructor of this object has four parameters that use can change.

Example1Proxy::Example1Proxy(int timeout = 5, int domainId = 0);

A brief description of each parameter:

* timeout: This parameter sets the time that each remote procedure call will wait its reply from server before sends the exception by timeout.
* domainId: This parameter sets the domain identifier that DDS will use.

The way to call a remote procedure is like calling a function in C++.

proxy->function1(p1, p2, p3, retValue);

Note: Rest of APIs return returned value as normal function. Think about this.

retValue = proxy->function1(p1, p2, p3);

### 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 DIL compiler. *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()*.

As RPC4DDS supports two type of transport (TCP or UDP), the IDL compiler generates two possible frontends for the user. *Example1WANServer* uses a *TCPTransport* object that allows to listen a port for TCP connections with clients. *Example1Server* uses a *UDPTransport* that uses the UDP discovery of DDS. Both proxies inherits from the class *Example1ServerH*. The *Example1ServerH* class inherits from the *Server* class. This class is generated by the IDL compiler 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 IDL compiler, 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 (Server)* 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()*.

*Server* can support several strategies when a request is received. *Server* is composed by an object which class inherits from the interface *ServerStrategy*. All strategies that will be implemented have to use this interface. User will be able to select the strategy when creates the server. There are the following strategies:

* OneThreadStrategy: The same thread that received the request will process it.
* ThreadPerRequestStrategy: For each new request, a new thread will be created and it will process the request.
* ThreadPoolStrategy: This strategy implements a thread pool. Each new request has to wait until a thread is free.

The IDL compiler 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.

### API

User has to create an object of the generated server to initialize the server. This generated server will expose the interface generated by the user. The constructor of this object has four parameters that use can change.

Example1Server::Example1Server(char \*listen\_addresses = NULL, StrategyServer \*strategy, int domainId = 0);

A brief description of each parameter:

* strategy: This parameter sets the strategy that server will use when a new request arrived.
* domainId: This parameter sets the domain identifier that DDS will use.

The generated server is disabled after its initialization. The user has to use the *run* function to execute the server.

Example1Server::run();

## Portability

This project needs to be run over several operations systems. There are objects and functionalities that depend on operations system. An example of these functionalities is: concurrency (mutex, conditions, and semaphores), threads. It is necessary a external library that implements this functionalities in all supported operation systems. There are several possibilities:

* ACE: <http://www.cs.wustl.edu/~schmidt/ACE.html>
* BOOST: <http://www.boost.org/>
* POCO: <http://pocoproject.org/>

ACE are focus in network, thought it implements concurrency and threads. But it hasn’t a thread pool in its implementation. BOOST is more general libraries and it exits a separate thread pool that uses BOOST. POCO is like boost, but it integrates a thread pool. The final decision is BOOST. Several reasons for this are:

* Some developers of BOOST are also developers of STL libraries. The future threads of STL are based in BOOST threads.
* BOOST is backed by a large community of developers.
* BOOST is used in more projects than the other libraries.
* BOOST is used in some projects that we’ve investigated.

# Behavior model

In this section is shown some functionality from sequence diagrams.

## Client

### Client initialization



### Client calling a remote procedure synchronously



### Client calling a remote procedure asynchronously



## Server

### Server initialization



### Server 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) .