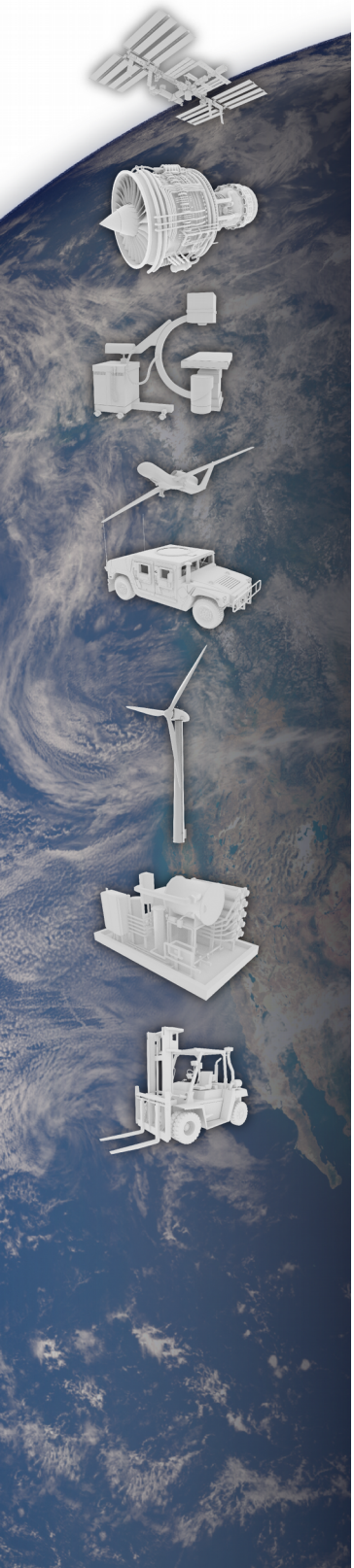

CoreDX DDS

RPC over DDS

Remote Procedure Call API

TWINOAKS COMPUTING INC.

PRACTICAL MIDDLEWARE EXPERTISE

2017-01-23

Table of Contents

Table of Figures

1 Introduction

CoreDX DDS v4 supports the OMG standard **Remote Procedure Call over DDS**. This technology provides a Remote Procedure Call (RPC) architecture that uses DDS for the underlying communication. The concepts are similar to other RPC architectures (such as CORBA or ONC RPC) with the ability to invoke a remote operation and optionally obtain a result. The combination of Publish-Subscribe and RPC in a single middleware infrastructure offers a powerful set of tools that address diverse system architecture requirements.

There are two fundamental API's offered by the RPC-over-DDS standard: the RequestReply API and the FunctionCall API. The RequestReply API is the 'low-level' API, while the FunctionCall API offers a more natural programming interface. The RequestReply API is made public because of some of the specific use-cases that it addresses that are not possible with the FunctionCall API; and, because it may be more comfortable in non-Object Oriented programming languages.

2 RPC over DDS Overview

The RPC over DDS architecture leverages existing DDS entities and protocols to support the Remote Procedure Call functionality. Each service is based on a DataReader / DataWriter pair for receiving requests and sending replies. Conversely, each client has a DataWriter / DataReader pair for sending requests and receiving replies. The Readers involved use ContentFilters to control the processing of specific requests and replies. The Service can filter on 'instanceName', if configured, to process requests intended only for a specific instance; and, the Client applies a filter so that only replies to locally originated requests are received.

The specifics of the applications interface to the RPC architecture is dependent on the selected language binding. The application can choose a Function-Call API, a Request-Reply API, or interact directly with the RPC DataReader[s] and DataWriter[s]. Figure 1 shows the high level architecture of RPC over DDS.

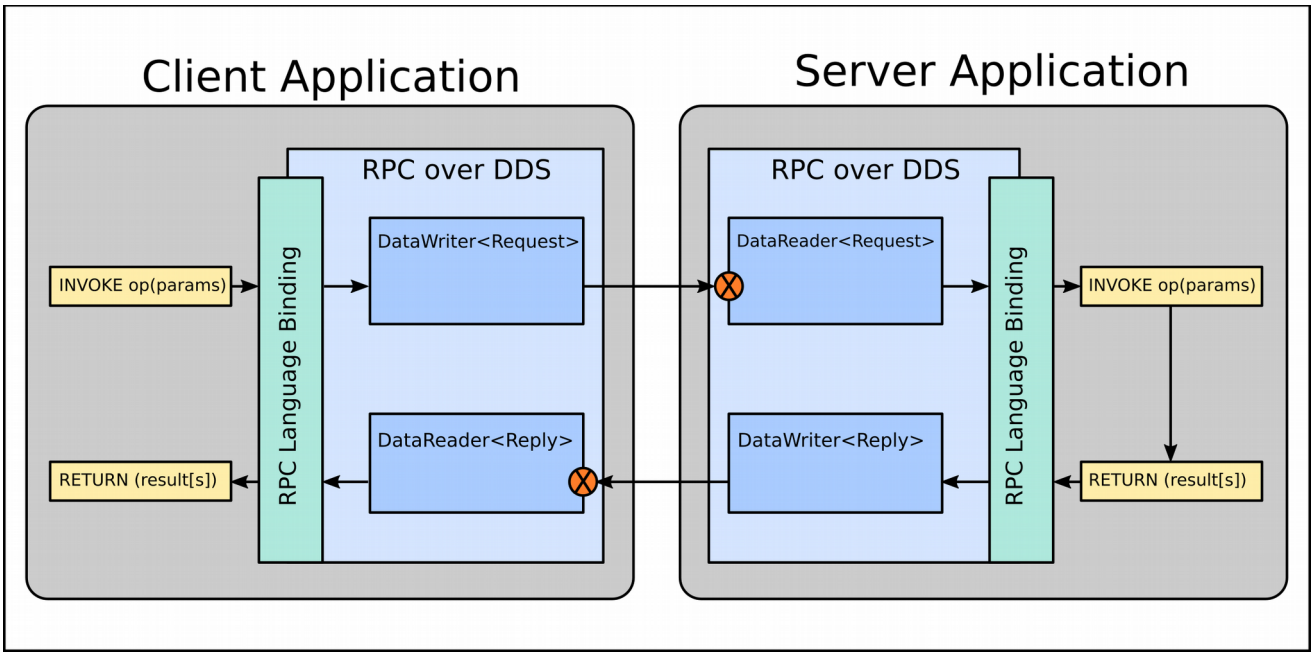

Figure 1: RPC over DDS Architecture

Because the RPC communication takes place over standard DDS DataReaders and DataWriters, all of the DDS communication facilities are available. For example, it is possible to invoke a single request that reaches multiple services; as such, a client can receive replies from all available Servers. Alternatively, a client could receive a stream of replies from one or more servers. These configurations are challenging to accomplish in more traditional RPC implementations. Of course, by binding a client to a specific service instance, it is possible to limit the request/reply transaction to a single Server.

The choice of language binding API (Request-Reply, Function-Call, raw) does not have to be consistent system-wide. That is, some clients may choose Request-Reply while others choose Function-Call. Further, the client language binding does not have to match the server language binding.

3 Interface Specification

An RPC Service is defined by an interface. An interface is a collection of one or more method declarations. An interface may derive from another interface, in order to extend the set of methods. The methods can accept parameters and generate a return value. The parameters can be declared as IN, OUT, or IN-OUT parameters, indicting that the parameter is used respectively as an input only, an output only, or an input and an output. Parameters and return values can be of any legal data type, including constructed types (struct, union, etc). A method can be declared to raise zero or more exceptions.

```
exception TooFast {};

enum Command { START_COMMAND, STOP_COMMAND };

struct Status
{
  string msg;
};

@service
interface RobotControl
 {
   void  command(Command com);
   float setSpeed(float speed) raises (TooFast);
   float getSpeed();
   void  getStatus(out Status status);
};
```

Figure 2: Example Interface Definition

Interfaces are specified in IDL with the 'interface' reserved word and annotated with the '@service' annotation to indicate that it is a DDS Service interface. For example, see Figure 2.

This IDL defines an interface named 'RobotControl' that includes four methods. One of the methods may raise a user defined exception 'TooFast'. One method has an 'out' parameter. The interface is annotated with '@service' to indicate that the IDL compiler should generate DDS-over-RPC support code for the interface.

Processing this IDL code with the CoreDX DDS IDL compiler (coredx_ddl) will generate source code that implements the datatypes necessary to deploy an RPC over DDS architecture. This generated code extends a standardized object model defined in the RPC over DDS specification. In order to understand and use the generated code, an application developer must also understand this standardized RPC object model.

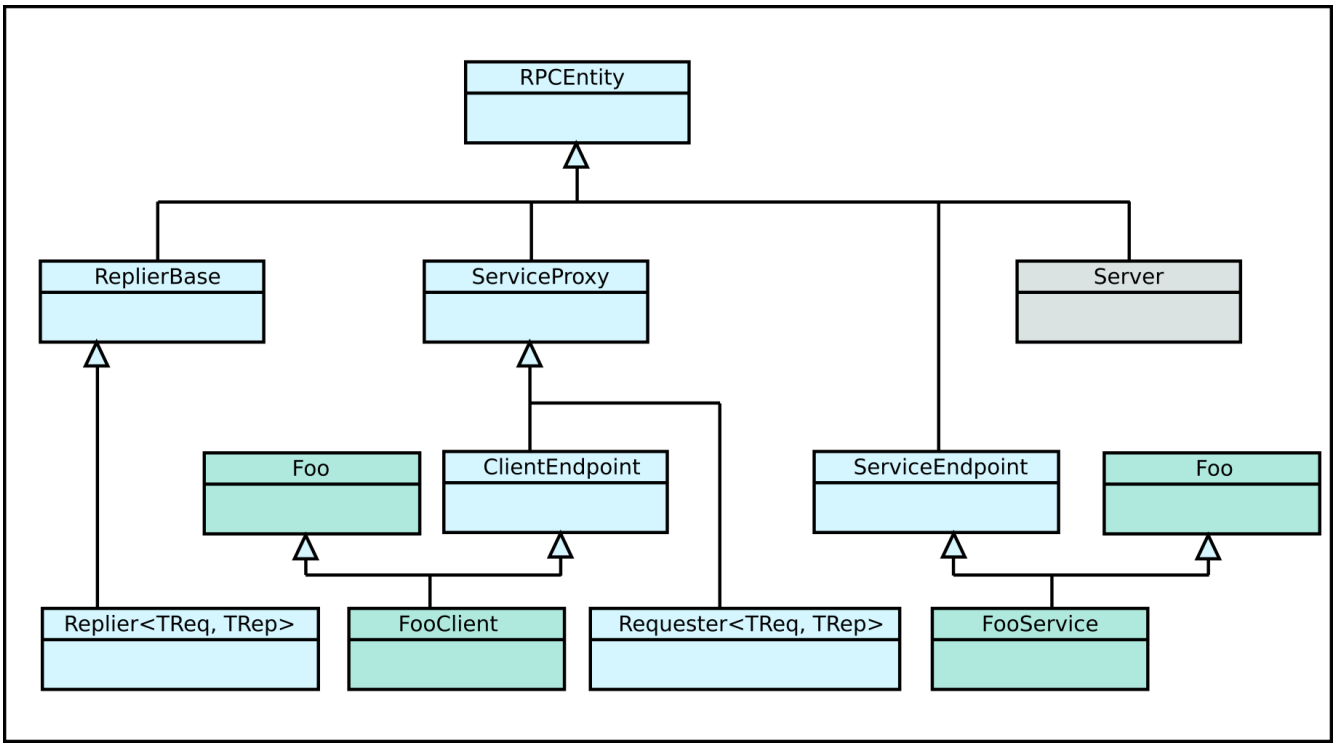

Figure 3: RPC Object Diagram

When using the Function Call API, the application developer will be interacting with object derived from the ClientEndpoint and ServiceEndpoint classes (for example, FooClient and FooService). When using the Request Reply API, the application developer will interact with objects derived from the Requester and Replier classes.

4 Function Call Interface

The interface definition maps to a '**client**' type and a '**service**' type.

The **client** type inherits from the **ClientEndpoint** class and the 'interface' class (eg, **Foo**) that adds the specific methods defined in the interface. An instance of the **client** type can be used by an application to make calls to the service.

The **service** type derives from the **ServiceEndpoint** class and the 'interface' class (eg, **Foo**). The 'service' must implement the pure-virtual service specific methods and create one or more instances of the **service** type.

4.1 FooClient

This figure shows the generated 'FooClient' class (for an interface named Foo). It includes the complete set of operations derived from the RPC base classes. The FooClient class is code generated from the IDL interface definition.

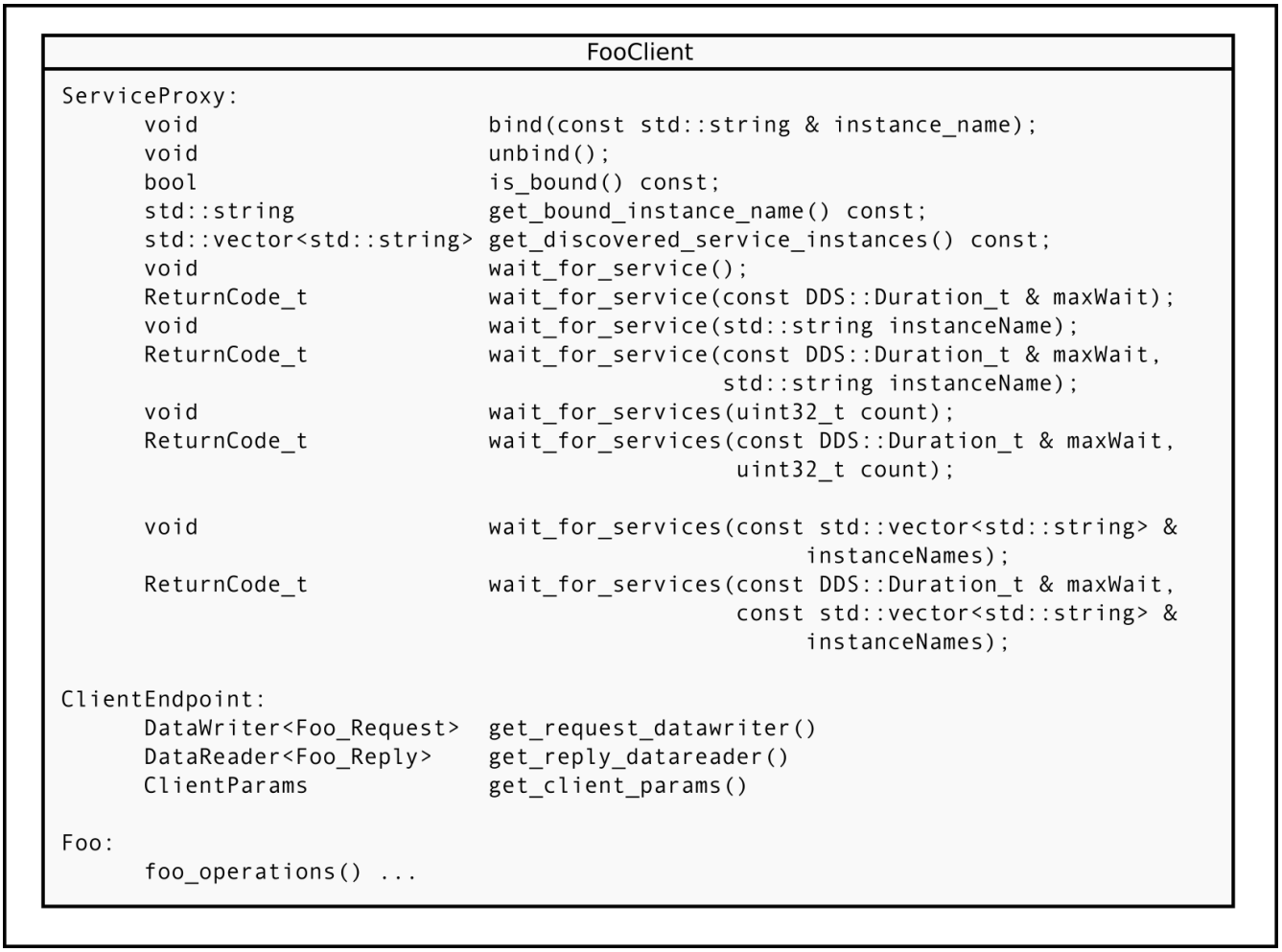

Figure 4: RPC Client Operations

4.1.1 FooClient Construction

The generated FooClient class includes a constructor that takes a parameter of type ClientParams. This ClientParams object contains a set of configuration parameters that are applied to the Client instance.

4.1.2 ClientParams

4.1.2.1 ClientParams::service_name

Specifies the service_name configured in this instance of ClientParams. The service_name, if specified, overrides the default service name “Service”.

4.1.2.2 ClientParams::instance_name

Specifies the instance_name configured in this instance of ClientParams. If the 'instance_name' is set, then it is used to 'bind()' the Client to a specific instance. See ServiceProxy::bind() for description.

4.1.2.3 ClientParams::request_topic_name

Specifies the request_topic_name configured in this instance of ClientParams. The request_topic_name, if set, is used to over-ride the default topic name used for the Request DataWriter.

4.1.2.4 ClientParams::reply_topic_name

Specifies the reply_topic_name configured in this instance of ClientParams. The reply_topic_name, if set, is used to over-ride the default topic name used for the Reply DataReader.

4.1.2.5 ClientParams::datawriter_qos

Specifies the DataWriterQos configured in this instance of ClientParams. This datawriter_qos, if set, is used when constructing the Request DataWriter.

4.1.2.6 ClientParams::datareader_qos

Specifies the DataReaderQos configured in this instance of ClientParams. The datareader_qos, if set, is used when constructing the Reply DataReader.

4.1.2.7 ClientParams::publisher

Specifies the Publisher configured in this instance of ClientParams. The publisher, if set, is used to create the Request DataWriter. If it is not set, then the Client will construct its own publisher instance.

4.1.2.8 ClientParams::subscriber

Specifies the Subscriber configured in this instance of ClientParams. The subscriber, if set, is used to create the Reply DataReader. If it is not set, then the Client will construct its own subscriber instance.

4.1.2.9 ClientParams::domain_participant

Specifies the DomainParticipant configured in this instance of ClientParams. The domain_participant, if set, will be used to construct the publisher, subscriber (if not also provided), and topics required for the internal DataReader and DataWriter.

4.1.3 FooClient Operations

The RPC infrastructure provides a number of operations that are available to a client through inheritance from ServiceProxy and ClientEndpoint. The following sections document those operations and their effect on the Client.

4.1.3.1 ServiceProxy::bind()

The bind() operation supports binding the client to a specific service 'instance'. The specific instance is identified by a string name. Once the bind() operation is called, all requests include the 'instanceName', and will be handled only by a Service with a matching name.

4.1.3.2 ServiceProxy::unbind()

The unbind() operation undoes a previous call to bind(). After an unbind() call, the Client is bound to no instance, and requests contain an empty string for the instanceName – all available services will receive and process the request.

4.1.3.3 ServiceProxy::is_bound()

The is_bound() operation returns a boolean indicating if the client is currently bound (has a non-empty instanceName configured).

4.1.3.4 ServiceProxy::get_bound_instance_name()

The get_bound_instance_name() returns the instanceName to which the Client is currently bound (if any).

4.1.3.5 ServiceProxy::get_discovered_service_instances()

Accesses a list of instance_names that have been discovered by the Client. The returned list is a snapshot in time of the currently known services.

4.1.3.6 ServiceProxy::wait_for_service()

This operation blocks until any service is matched. The caller can specify a maximum duration or a specific instanceName, or both. If no instanceName is specified, but the client is currently bound, then the currently bound instanceName is used.

4.1.3.7 ServiceProxy::wait_for_services()

This operation blocks until at least a certain number of services are discovered. The caller can specify a maximum blocking duration, and a list of instanceNames.

4.1.3.8 ClientEndpoint::get_request_datawriter()

Accesses the DataWriter that is used to send outgoing requests.

4.1.3.9 ClientEndpoint::get_reply_datareader()

Access the DataReader that is used to accept incoming replies.

4.1.3.10 ClientEndpoint::get_client_params()

Access the clientParameters that were provided to the constructor of the Client. These parameters are used to configure various properties of the client.

4.2 FooService

This figure shows the generated 'FooService' class (for an interface named 'Foo'). It includes the complete set of operations derived from the RPC base classes.

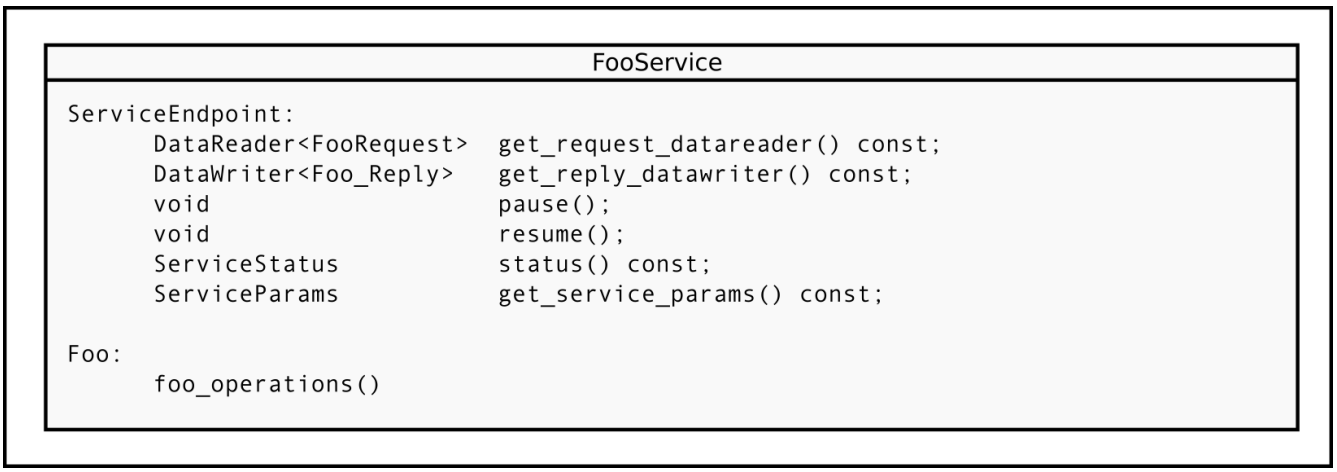

Figure 5: RPC Service Operations

4.2.1 FooServiceConstruction

The generated FooService code includes a constructor that accepts a parameter of type ServiceParams.

4.2.1.1 ServiceParams

ServiceParams is a collection of configuration items to control the construction and behavior of a Service instance.

4.2.1.1.1 ServiceParams::service_name

Specifies the service_name configured in this instance of ServiceParams. The service_name, if specified, overrides the default service name “Service”.

4.2.1.1.2 ServiceParams::instance_name

Specifies the instance_name configured in this instance of ServiceParams. If the 'instance_name' is set, then service instance is tied to a specific instance.

4.2.1.1.3 ServiceParams::request_topic_name

Specifies the request_topic_name configured in this instance of ServiceParams. The request_topic_name, if set, is used to over-ride the default topic name used for the Request DataReader.

4.2.1.1.4 ServiceParams::reply_topic_name

Specifies the reply_topic_name configured in this instance of ServiceParams. The reply_topic_name, if set, is used to over-ride the default topic name used for the Reply DataWriter.

4.2.1.1.5 ServiceParams::datawriter_qos

Specifies the DataWriterQos configured in this instance of ServiceParams. This datawriter_qos, if set, is used when constructing the Reply DataWriter.

4.2.1.1.6 ServiceParams::datareader_qos

Specifies the DataReaderQos configured in this instance of ServiceParams. The datareader_qos, if set, is used when constructing the Request DataReader.

4.2.1.1.7 ServiceParams::publisher

Specifies the Publisher configured in this instance of ServiceParams. The publisher, if set, is used to create the Reply DataWriter. If it is not set, then the Service will construct its own publisher instance.

4.2.1.1.8 ServiceParams::subscriber

Specifies the Subscriber configured in this instance of ServiceParams. The subscriber, if set, is used to create the Request DataReader. If it is not set, then the Service will construct its own subscriber instance.

4.2.1.1.9 ServiceParams::domain_participant

Specifies the DomainParticipant configured in this instance of ServiceParams. The domain_participant, if set, will be used to construct the publisher, subscriber (if not also provided), and topics required for the internal DataReader and DataWriter.

4.2.2 Service Operations

The RPC infrastructure provides a number of methods to the Service through inheritance from ServiceEndpoint.

4.2.2.1 ServiceEndpoint::get_request_datareader()

Access the DataReader that is used to accept incoming requests.

4.2.2.2 ServiceEndpoint::get_reply_datawriter()

Access the DataWriter that is used to send outgoing replies.

4.2.2.3 ServiceEndpoint::pause()

The pause() operation pauses the operation of the ServiceEndpoint. This causes the ServiceEndpoint to not invoke an attached listener.

4.2.2.4 ServiceEndpoint::resume()

The resume() operation resumes the operation of the ServiceEndpoint. This causes the ServiceEndpoint to invoke any attached listeners upon reception of a request. In order to handle any requests received while the ServiceEndpoint was paused, the listener will be invoked as part of the resume() operation.

4.2.2.5 ServiceEndpoint::status()

Access the current status of the ServiceEndpoint. This will return one of CLOSED, PAUSED, or RUNNING.

4.2.2.6 ServiceEndpoint::get_service_params()

Access the ServiceParams defining the configuration of this ServiceEndpoint.

4.3 RobotControl Example (Function Call)

For example, the 'RobotControl' interface definition presented above will generate a **RobotControlClient** and **RobotControlService**.

Here is the generated RobotControlClient class:

```
struct COREDX_TS_STRUCT_EXPORT RobotControlClient :
  public RobotControl,
  public DDS::rpc::ClientEndpoint {
  public:
    /** Constructor, Copy Constructor, Destructor, Assignment operator */
    RobotControlClient(DDS::rpc::ClientParams & params);
    RobotControlClient( const RobotControlClient & other );
    virtual ~RobotControlClient();
    RobotControlClient & operator=( const RobotControlClient & other);
    void                 command ( /* IN    */  const enum robot::Command   com );
    float                setSpeed ( /* IN    */  const float  speed );
    float                getSpeed ( );
    void                 getStatus ( /* OUT   */  struct robot::Status & status );
  protected:
     coredx::rpc::Requester * requester;
 }; //robot::RobotControlClient
```

Figure 6: Example RobotControl Client

And here is the generated RobotControlService class:

```
struct COREDX_TS_STRUCT_EXPORT RobotControlService :
  public RobotControl,
  public DDS::rpc::ServiceEndpoint {
  public:
    /** Constructor, Copy Constructor, Destructor, Assignment operator */
    RobotControlService(DDS::rpc::ServiceParams & params);
    RobotControlService( const RobotControlService & other );
    virtual ~RobotControlService();
    RobotControlService & operator=( const RobotControlService & other);
    class RobotControlServiceListener : public DDS::DataReaderListener
    {
      public:
       RobotControlServiceListener(RobotControlService * service);
       virtual void on_data_available(DDS::DataReader* the_reader);
       RobotControlService * the_service;
    };
    virtual void                command (/* IN */  const enum robot::Command com ) = 0;
    virtual float               setSpeed (/* IN */  const float  speed ) = 0;
    virtual float               getSpeed () = 0;
    virtual void                getStatus (/*OUT*/  struct robot::Status &status) = 0;
    void   dispatch(RobotControl_Request &req);
  protected:
    DDS::rpc::Replier<RobotControl_Request, RobotControl_Reply> * replier;
}; //robot::RobotControlService
```

Figure 7: Example RobotControl Service

The application must derive a class from RobotControlService, and provide an implementation for the pure-virtual methods. The implementation of the 'service' could look like this:

```
// ---------------------------------------------------------------------------

/* RobotControlService : application logic  -- user written  */
class MyRobotControlService : public robot::RobotControlService
{
  static const float MAX_SPEED;

public:

  MyRobotControlService(DDS::rpc::ServiceParams & params) :
    RobotControlService(params),
    current_speed(0.0),
    current_status("STOPPED")
  {
  }

  MyRobotControlService( const RobotControlService & other ) :
    RobotControlService(other)
  {
  }
```

```
  virtual ~MyRobotControlService() {}
  RobotControlService & operator=( const RobotControlService & other){}

  void
  command ( /* IN    */  const enum robot::Command   com )
  {
    switch (com)
      {
      case robot::START_COMMAND:
        this->current_status = "RUNNING";
        break;
      case robot::STOP_COMMAND:
        this->current_status = "STOPPED";
        break;
      default:
        printf("Unknown 'command' value...\n");
        /* operation could be declared to throw exception */
        break;
      }
  }

  float
  setSpeed ( /* IN    */  const float  speed )
  {
    float retval = 0.0;
    if (speed < MAX_SPEED)
      {
        current_speed = speed;
        retval = this->current_speed;
      }
    else
      {
        /* EXCEPTION: */
        robot::TooFast tooFast;
        throw tooFast;
        /* not reached... */
      }
    return retval;
  }

  float
  getSpeed ( )
  {
    return this->current_speed;
  }

  void
  getStatus ( /* OUT   */  struct robot::Status & status )
  {
    status.msg = new char[strlen(current_status)+1];
    strcpy(status.msg, this->current_status);
  }

protected:
  float        current_speed;
  const char * current_status;
};
```

```
const float MyRobotControlService::MAX_SPEED = 20.0;
```

Figure 8: Example RobotControl Service code

The server must create one or more instances of this Service Implementation:

```
robot::RobotControlService * robotService;
DDS::rpc::ServiceParams      serviceParams;

serviceParams.
  domain_participant(participant).
  service_name(“RobotControl”).
  instance_name(“myRobot”);

robotService = new MyRobotControlService( serviceParams );
```

Figure 9: Example: Creating a RobotControlService Instance

Each instance can be given a specific 'instance_name' so that a client can 'bind' to it specifically.

At this point, the service is created and running, ready to receive and respond to method invocations.

The client application code instantiates an instance of RobotControlClient, and makes calls on that object. The client invocation could look something like this:

```
robot::RobotControlClient * robotClient;
DDS::rpc::ClientParams      clientParams;
DDS::DomainParticipant    * participant;
robot::Status               robotStatus;

clientParams.
  domain_participant(participant).
  service_name(“RobotControl”);

robotClient = new robot::RobotControlClient( clientParams );

robotClient->bind(“myRobot”);
robotClient->wait_for_service();
robotClient->getStatus( robotStatus );
// handle 'robotStatus' . . .
```

Figure 10: Example RobotControl Client usage

5 Request Reply Interface

The Request Reply interface provides more fine-grained control over the messaging between the 'client' (Requester) and 'service' (Replier). This level of interface requires that the application deal with more complex data-types and rules.

One benefit of this approach is the ability to receive multiple replies in response to a single request. Further, the application has multiple options for how it accesses Requests and Replies. It can use polling, blocking, or asynchronous listener callbacks for notification of the presence of Request and Reply data. Then, it can process these one at a time, or as a set; and it can choose to either take or read the samples.

5.1 Requester<TReq, TReq>

The Requester is a generic class that is specialized with the Request and Reply data types. The application will instantiate a specialized instance of this class, and then utilize the available operations to send requests and receive replies.

5.1.1 Requester Construction

A Requester has two template parameters: the Request type and the Reply type. The constructor requires a parameter of type RequesterParams that provides a collection of configuration parameters.

5.1.1.1 RequesterParams

The RequesterParams object is a container of several configuration parameters. The object includes operations to get and set the various parameters. The parameters and their meanings are documented here.

5.1.1.1.1 RequesterParams::simple_requester_listener

This parameter assigns the 'simple' listener that will be installed in the Requester. Only one listener may be installed, either the SimpleListener or RequesterListener. It is valid to leave both listener parameters unassigned, in which case no listener will be installed.

5.1.1.1.2 RequesterParams::requester_listener

This parameter assigns the 'requester' listener that will be installed in the Requester. Only one listener may be installed, either the SimpleListener or RequesterListener. It is valid to leave both listener parameters unassigned, in which case no listener will be installed.

5.1.1.1.3 RequesterParams::domain_participant

This parameter assigns the DomainParticipant to use. The requester will use this DomainParticipant to create any required Publisher, Subscriber, DataWriter and DataReader entities. If the

domain_participant is left unassigned, then the Requester will create its own DomainParticipant.

5.1.1.1.4 RequesterParams::publisher

This parameter assigns the Publisher to use. The Requester will use this publisher to create any required DataWriter entities. If not provided the Requester will create its own Publisher.

5.1.1.1.5 RequesterParams::subscriber

This parameter assigns the Subscriber to use. The Requester will use this subscriber to create any requried DataReader entities. If not provided, the Requester will create its own Subscriber.

5.1.1.1.6 RequesterParams::datawriter_qos

This parameter assigns the DataWriterQos to use. This QoS will be used when the Requester creates a DataWriter.

5.1.1.1.7 RequesterParams::datareader_qos

This parameter assigns the DataReaderQos to use. This QoS will be used when the Requester creates a DataReader.

5.1.1.1.8 RequesterParams::service_name

This parameter assigns the service_name to use when creating the Request and Reply topic names. The service_name is used when constructing the topic name used for the Request and Reply topics. If not explicitly defined by the request_topic_name() parameter, the request topic is composed of <service_name>_Request. Similarly, if not overridden by reply_topic_name(), the reply topic name is composed of <service_name>_Reply. If the service_name() item is left unspecified, then the string "Service" is used.

5.1.1.1.9 RequesterParams::request_topic_name

This parameter assigns the 'request_topic_name' to use as the Request topic name. This overrides the default topic name generation.

5.1.1.1.10 RequesterParams::reply_topic_name

This parameter assigns the 'reply_topic_name' used as the Reply topic name. This overrides the default topic name generation.

5.1.2 Requester Listeners

A Requester can be configured with a single listener. There are two types of listeners that can be configured: the SimpleRequesterListener and the RequesterListener. These two listeners vary in how they interface to the application.

5.1.2.1 SimpleRequesterListener

SimpleRequesterListener is a generic class with a template parameter of the Reply type. It includes one abstract operation that must be implemented by the application.

```
void process_reply(const Sample<TRep> &,
                   const DDS::SampleIdentity_t &) = 0;
```

The process_reply() operation is called each time a reply is received by the Requester. The details of the reply (the Reply data type) and the SampleIdentity of the original request are provided as parameters.

5.1.2.2 RequesterListener

RequesterListener is a generic class with two template parameters: the Request type and the Reply type. It includes a single abstract operation that must be implemented by the application.

```
void on_reply_available(Requester<TReq, TRep> &) = 0;
```

The on_reply_available() operation is called when it is detected that one or more replies are available. The application can then access the reply/replies through the receive/read/take operations on the Requester.

5.1.3 Requester Operations

A Requester sends requests and receives replies. An instance of a Requester is configured at the time of construction using RequesterParams, which is a container of configuration parameters, such as domain participant, QoS, listeners and more.

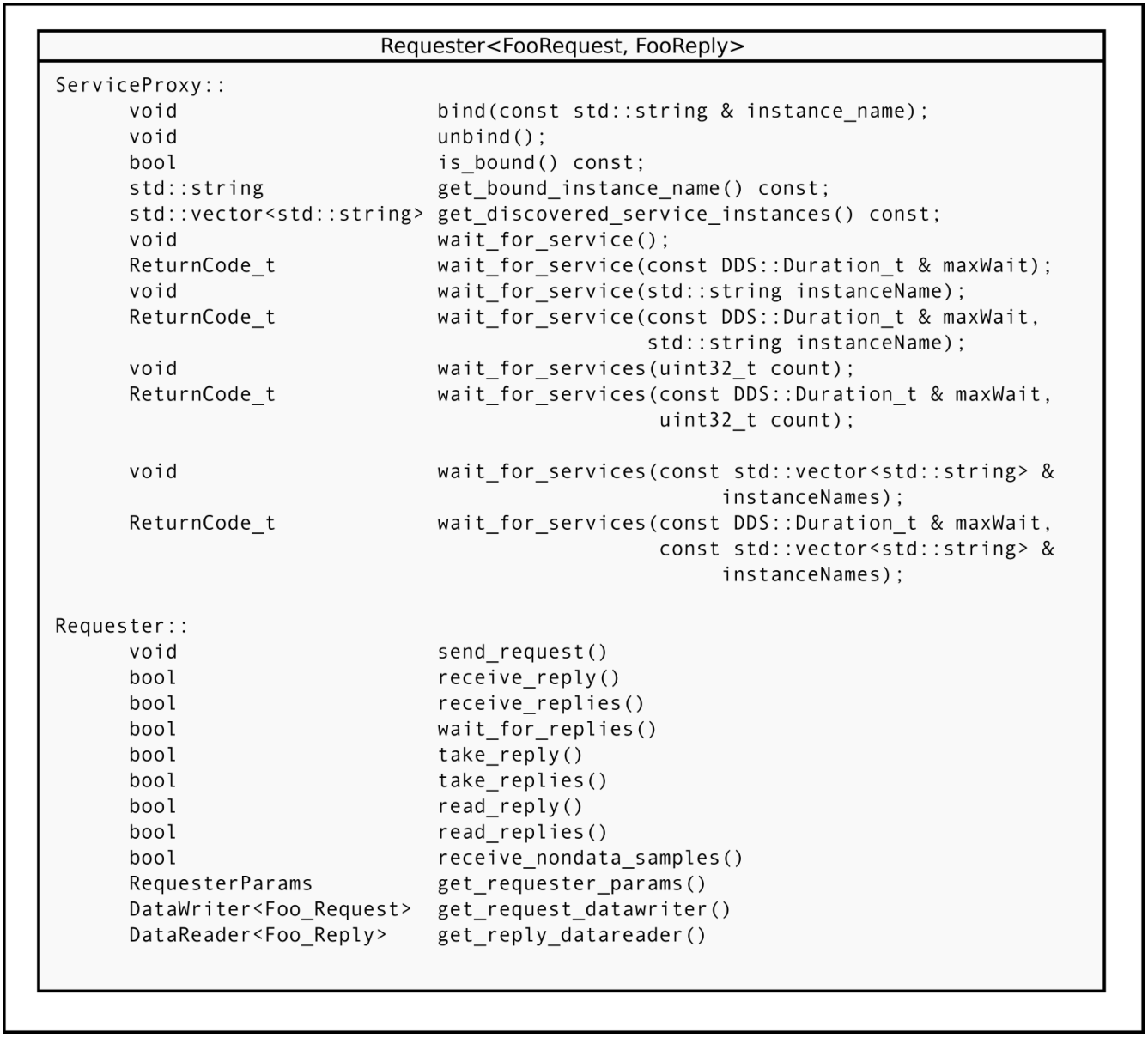

Figure 11: RPC Requester Operations

Requester is inherently asynchronous as sending a request and receiving its corresponding reply (or replies) are separated. Requester allows listener-based, polling-based, and future-based reception of replies.

SimpleRequesterListener and RequesterListener interfaces enable callback-based notification when a reply is available. On the other hand, Requester provides functions to allow polling reception of replies.

Future-based notification of replies is analogous to callback-based notification, however, no request-

reply correlation is necessary because every future represents a reply to a unique request. A requester reference may be bound to a specific service instance. Requests sent through a bound requester reference shall be sent to the bound service instance only.

The Requester includes operations from ServiceProxy. These operations are documented in Section 4.1.3 FooClient Operations. The additional operations provided by Requester itself are documented here.

5.1.3.1 Requester::send_request(),

This operation transmits a request to any matched instances of the service.

5.1.3.2 Requester::receive_reply(), Requester::receive_replies()

The receive_reply() operations attempt to access one or more received replies. These methods all accepts a 'timeout' parameter indicating how long they should block if no replies are immediately available. If one or more replies are accessed, then the operations will return true; otherwise, they return false. Some variations of these operations accept a 'relatedRequestId' parameter. The 'relatedRequestId' parameter limits the considered replies to only those that are in response to the specified request.

5.1.3.3 Requester::wait_for_replies()

The wait_for_replies() operation blocks until one or more replies are available. If no replies are currently available, the routine will block for up to 'max_wait'. Variations of this operation accept an integer 'min_count' requesting that the operation blocks until at least 'min_count' replies are available. If the specified replies are available, the operation will return true; otherwise, false.

5.1.3.4 Requester::take_reply(), Requester::take_replies()

The take_reply() operations operate like receive_reply() exception that they will not block.

5.1.3.5 Requester::read_reply(), Requester::read_replies()

The read_reply() operations behave like receive_reply() except that they will always block until the specified replies are available.

5.1.3.6 Requester::receive_nondata_samples()

Toggle whether non-data samples are presented to the application. This is of little utility, as the only non-data samples are unregister, dispose, and not-alive indications which are uncommon with an un-keyed data type.

5.1.3.7 Requester::get_requester_params()

Accesses the RequesterParameters provided when constructing this Requester.

5.1.3.8 Requester::get_request_datawriter()

Provides access to the underlying DataWriter used for sending Requests.

5.1.3.9 Requester::get_reply_datareader()

Provides access to the underlying DataReader used for receiving Replies.

5.2 Replier<TReq, TRep>

The RPC Replier is a generic class that is specialized with the Request and Reply data types. The application code will instantiate a specialized instance of the Replier, and then utilize the available operations to read requests and send replies.

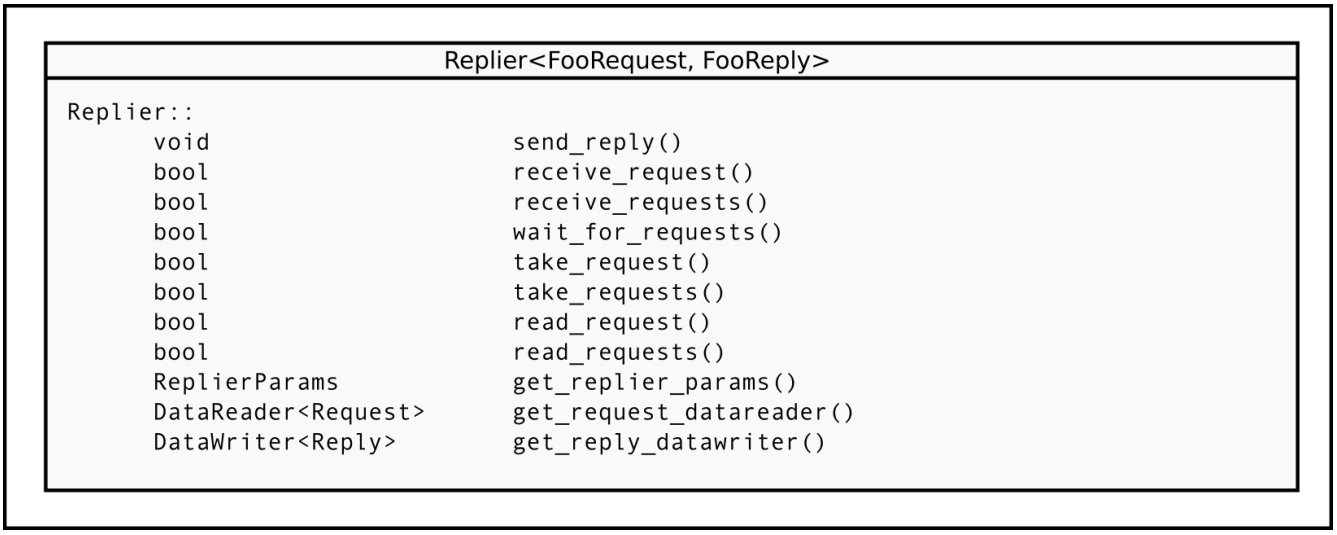

Figure 12: RPC Replier Operations

5.2.1 Replier Construction

A Replier has two template parameters: the Request type and the Reply type. The constructor requires a parameter of type ReplierParams that provides a collection of configuration parameters.

5.2.1.1 ReplierParams

The ReplierParams object is a container of several configuration parameters. The object includes operations to get and set the various parameters. The parameters and their meanings are documented here.

5.2.1.2 ReplierParams::simple_replier_listener

Specify the SimpleReplierListener configured in this instance of ReplierParams. Only one listener may

be installed, either the SimpleReplierListener or ReplierListener. It is valid to leave both listener parameters unassigned, in which case no listener will be installed.

5.2.1.3 ReplierParams::replier_listener

Specify the ReplierListener configured in this instance of ReplierParams. Only one listener may be installed, either the SimpleReplierListener or ReplierListener. It is valid to leave both listener parameters unassigned, in which case no listener will be installed.

5.2.1.4 ReplierParams::domain_participant

Specify the DomainParticipant configured in this instance of ReplierParams. The replier will use this DomainParticipant to create any required Publisher, Subscriber, DataWriter and DataReader entities. If the domain_participant is left unassigned, then the Replier will create its own DomainParticipant.

5.2.1.5 ReplierParams::service_name

Specify the service_name configured in this instance of ReplierParams. The service_name is used when constructing the topic name used for the Request and Reply topics. If not explicitly defined by the request_topic_name() parameter, the request topic is composed of <service_name>_Request. Similarly, if not overridden by reply_topic_name(), the reply topic name is composed of <service_name>_Reply. If the service_name() item is left unspecified, then the string "Service" is used.

5.2.1.6 ReplierParams::instance_name

Specify the instance_name configured in this instance of ReplierParams. The service will advertise this 'instance_name' so that Clients can 'bind' to a particular instance.

5.2.1.7 ReplierParams::request_topic_name

This parameter assigns the 'request_topic_name' to use as the Request topic name. This overrides the default topic name generation.

5.2.1.8 ReplierParams::reply_topic_name

This parameter assigns the 'reply_topic_name' used as the Reply topic name. This overrides the default topic name generation.

5.2.1.9 ReplierParams::datawriter_qos

Specify the DataWriterQos configured in this instance of ReplierParams. This QoS will be used when the Replier creates a DataWriter.

5.2.1.10 ReplierParams::datareader_qos

Specify the DataReaderQos configured in this instance of ReplierParams. This QoS will be used when the Replier creates a DataReader.

5.2.1.11 ReplierParams::publisher

Specify the Publisher configured in this instance of ReplierParams. The Replier will use this publisher to create any required DataWriter entities. If not provided the Replier will create its own Publisher.

5.2.1.12 ReplierParams::subscriber

Specify the Subscriber configured in this instance of ReplierParams. The Replier will use this subscriber to create any requried DataReader entities. If not provided, the Replier will create its own Subscriber.

5.2.2 Replier Listeners

A Replier can be configured with a single listener. There are two types of listeners that can be configured: the SimpleReplierListener and the ReplierListener. These two listeners vary in how they interface to the application.

5.2.2.1 SimpleReplierListener

SimpleReplierListener is a generic class with two template parameters of the Request type and the Reply type. It includes one abstract operation that must be implemented by the application.

```
Reply process_request(const Sample<Request> &,
                      const DDS::SampleIdentity_t &) = 0;
```

The process_request() operation is called each time a request is received by the Replier. The details of the request (the Request data type) and the SampleIdentity of the request are provided as parameters. The operation should construct an instance of Reply type initialized appropriately and return that object. The RPC infrastructure will ensure that the reply is transmitted to the requester.

5.2.2.2 ReplierListener

ReplierListener is a generic class with two template parameters of the Request type and the Reply type. It includes one abstract operation that must be implemented by the application.

```
void on_request_available(Replier<Request, Reply> &) = 0;
```

The on_request_available() method can process the available requests and send replies as required. This operation must utilize existing Replier methods to access requests and to transmit replies.

5.2.3 Replier Operations

An instance of Replier is configured at the time of construction using ReplierParams, which is a container of configuration parameters such as domain participant, QoS, listeners and more.

SimpleReplierListener and ReplierListener interfaces enable call-back based notification when a request is available. On the other hand, Replier provides functions to allow polling reception of requests.

5.2.3.1 Replier::send_reply()

Sends the provided 'reply' associated with 'related_request_id'. The caller provides the Reply data type along with the SampleIdentity taken from a previously received request (the related_request_id).

5.2.3.2 Replier::receive_request(), Replier::receive_requests()

These operations attempt to access one or more requests. They can be provided a Duration specifying the maximum amount of time to block before returning if the request[s] are not immediately available. Variations of these methods accept the minimum and maximum request count. These operations will block until at least one request is available or until the maximum blocking duration has elapsed.

5.2.3.3 Replier::wait_for_requests()

The wait_for_requests() operations block until some number of requests are available or until a specific blocking duration elapses. Variations of this operation are available to accept different termination conditions.

5.2.3.4 Replier::take_request(), Replier::take_requests()

The take_request() operations are similar to the receive_request() operations except that they will not ever block.

5.2.3.5 Replier::read_request(), Replier::read_requests()

The read_request() operations are similar to receive_request() except that they will always block until the request[s] are available.

5.2.3.6 Replier::get_replier_params()

Accesses the ReplierParams provided when initializing the Replier.

5.2.3.7 Replier::get_request_datareader()

Accesses the underlying DataReader used to receive Requests.

5.2.3.8 Replier::get_reply_datawriter()

Accesses the underlying DataWriter used to send Replies.

5.3 Parameter and Return Value Mapping

The parameters and return value (if present) of each method are collected into a pair of datatypes: the

"_In" and "_Out" types. That is, for each method declared in the interface, one _In type and one _Out type is defined. Then, all _In types are collected into a "_Call" type, and all _Out types are collected into a "_Return" type. The _Call and _Return types are a union of all _In and all _Out types, respectively. Finally, these _Call and _Return types are wrapped into the "_Request" and "_Reply" types. These data types are used in the RequestReply API, and are used internally to implement the FunctionCall API.

When using the RequestReply API, the developer must interact with these generated data types. **When using the FunctionCall API, the same data types are used internally; however, the developer does not interact with them directly.**

5.4 RobotControl Example (Request-Reply)

The following text shows some generated code to illustrate how parameters and operations are mapped to datatypes.

For example, the 'RobotControl::setSpeed' operation input parameters are collected in a datatype named RobotControl_setSpeed_In. The generated code for this datatype is shown below. Most of this is boiler-plate code used by the CoreDX DDS infrastructure. The item of interest is the 'float speed;' element which corresponds to the single parameter to the setSpeed operation.

```
struct COREDX_TS_STRUCT_EXPORT RobotControl_setSpeed_In {
   public:
     /** Constructor, Copy Constructor, Destructor, Assignment operator */
     RobotControl_setSpeed_In();
     ~RobotControl_setSpeed_In();
     RobotControl_setSpeed_In( const RobotControl_setSpeed_In & other );
     RobotControl_setSpeed_In& operator=( const RobotControl_setSpeed_In & other);

     void init();
     void clear();
     void copy( const robot::RobotControl_setSpeed_In * instance );

     int  get_marshal_size(int offset, int just_keys) const ;
     int  marshal_cdr(unsigned char * buf, int offset, int stream_len,
                 unsigned char swap, int just_keys) const ;
     int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
     int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len,
                 unsigned char swap, int just_keys);
     int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

     /* Member vars*/
     float  speed;

     typedef RobotControl_setSpeed_InTypeSupport TypeSupport;
```

```
      typedef RobotControl_setSpeed_InDataReader DataReader;
      typedef RobotControl_setSpeed_InDataWriter DataWriter;
      typedef RobotControl_setSpeed_InPtrSeq Seq;

   private:

}; //robot::RobotControl_setSpeed_In
```

Figure 13: Example RobotControl setSpeed _In

The RobotControl::setSpeed() _Out datatype is named RobotControl_setSpeed_Out, and looks like the following. Again, the important element is the “float _return;” member.

```
struct COREDX_TS_STRUCT_EXPORT RobotControl_setSpeed_Out {
   public:
     /** Constructor, Copy Constructor, Destructor, Assignment operator */
     RobotControl_setSpeed_Out();
     ~RobotControl_setSpeed_Out();
     RobotControl_setSpeed_Out( const RobotControl_setSpeed_Out & other );
     RobotControl_setSpeed_Out& operator=( const RobotControl_setSpeed_Out & other);

     void init();
     void clear();
     void copy( const robot::RobotControl_setSpeed_Out * instance );

     int  get_marshal_size(int offset, int just_keys) const ;
     int  marshal_cdr(unsigned char * buf, int offset, int stream_len,
                      unsigned char swap, int just_keys) const ;
     int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
     int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len,
                      unsigned char swap, int just_keys);
     int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

     /* Member vars*/
     float  return_;

     typedef RobotControl_setSpeed_OutTypeSupport TypeSupport;
     typedef RobotControl_setSpeed_OutDataReader DataReader;
     typedef RobotControl_setSpeed_OutDataWriter DataWriter;
     typedef RobotControl_setSpeed_OutPtrSeq Seq;

   private:

 }; //robot::RobotControl_setSpeed_Out
```

Figure 14: Example RobotControl setSpeed_Out

The _Call, _Return, _Request, and _Reply types generated for the RobotControl interface are shown

below. Again, most of the content of these datatypes is simply boiler-plate required by the infrastructure. Because these datatypes are essentially 'unions', they include a discriminator() method and accessor and modifier methods for each of their possible contained values.

```
class RobotControl_Call {  // robot::
  public:
    int  _discriminator;
    unsigned char  _initialized;
    struct robot::RobotControl_command_In  _pd_command;
    struct robot::RobotControl_setSpeed_In  _pd_setSpeed;
    struct robot::RobotControl_getSpeed_In  _pd_getSpeed;
    struct robot::RobotControl_getStatus_In  _pd_getStatus;
  public:
    // Constructor, Copy Constructor, Destructor, Assignment operator
    //  ... removed for brevity ...
    void  discriminator(int d)  { _discriminator = d; _initialized = 1; }
    int   discriminator() const      { return _discriminator; }

    void init();
    void clear();
    void copy( const RobotControl_Call * instance );

    int  get_marshal_size(int offset, int just_keys) const ;
    int  marshal_cdr(unsigned char * buf, int offset, int stream_len,
                     unsigned char swap, int just_keys) const ;
    int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
    int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len,
                unsigned char swap, int just_keys);
    int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

    /* Member vars*/
    const  struct robot::RobotControl_command_In &command() const
                { return _pd_command; }
     struct robot::RobotControl_command_In &command()
                { return _pd_command; }
    void command( const  struct robot::RobotControl_command_In &  _v) {
      clear();
      _pd_command = _v;
      this->discriminator(robot::RobotControl_command_hash);
    }
    const  struct robot::RobotControl_setSpeed_In &setSpeed() const
                { return _pd_setSpeed; }
     struct robot::RobotControl_setSpeed_In &setSpeed()
                { return _pd_setSpeed; }
    void setSpeed( const  struct robot::RobotControl_setSpeed_In &  _v) {
      clear();
      _pd_setSpeed = _v;
      this->discriminator(robot::RobotControl_setSpeed_hash);
    }
    const  struct robot::RobotControl_getSpeed_In &getSpeed() const
                { return _pd_getSpeed; }
     struct robot::RobotControl_getSpeed_In &getSpeed()
                { return _pd_getSpeed; }
    void getSpeed( const  struct robot::RobotControl_getSpeed_In &  _v) {
```

```
        clear();
        _pd_getSpeed = _v;
        this->discriminator(robot::RobotControl_getSpeed_hash);
      }
      const  struct robot::RobotControl_getStatus_In &getStatus() const
                 { return _pd_getStatus; }
       struct robot::RobotControl_getStatus_In &getStatus()
                 { return _pd_getStatus; }
      void getStatus( const  struct robot::RobotControl_getStatus_In &  _v) {
        clear();
        _pd_getStatus = _v;
        this->discriminator(robot::RobotControl_getStatus_hash);
      }
  };
```

Figure 15: Example RobotControl getStatus_Call

```
  class RobotControl_Return {  // robot::
    public:
      int  _discriminator;
      unsigned char  _initialized;
      union {
         DDS::rpc::UnknownOperation  _pd_unknownOp;
      } _u;
      class robot::RobotControl_command_Result  _pd_command;
      class robot::RobotControl_setSpeed_Result  _pd_setSpeed;
      class robot::RobotControl_getSpeed_Result  _pd_getSpeed;
      class robot::RobotControl_getStatus_Result  _pd_getStatus;
    public:
      // Constructor, Copy Constructor, Destructor, Assignment operator
      // ... removed for brevity ...
      void discriminator(int d)  { _discriminator = d; _initialized = 1; }
      int  discriminator() const      { return _discriminator; }

      void init();
      void clear();
      void copy( const RobotControl_Return * instance );

      int  get_marshal_size(int offset, int just_keys) const ;
      int  marshal_cdr(unsigned char * buf, int offset, int stream_len,
                 unsigned char swap, int just_keys) const ;
      int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
      int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len,
                 unsigned char swap, int just_keys);
      int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

      /* Member vars*/
      const  class robot::RobotControl_command_Result &command() const
                 { return _pd_command; }
       class robot::RobotControl_command_Result &command()
                 { return _pd_command; }
      void command( const  class robot::RobotControl_command_Result &  _v) {
        clear();
        _pd_command = _v;
```

```
        this->discriminator(robot::RobotControl_command_hash);
      }
      const  class robot::RobotControl_setSpeed_Result &setSpeed() const
                  { return _pd_setSpeed; }
       class robot::RobotControl_setSpeed_Result &setSpeed()
                  { return _pd_setSpeed; }
      void setSpeed( const  class robot::RobotControl_setSpeed_Result &  _v) {
        clear();
        _pd_setSpeed = _v;
        this->discriminator(robot::RobotControl_setSpeed_hash);
      }
      const  class robot::RobotControl_getSpeed_Result &getSpeed() const
                  { return _pd_getSpeed; }
       class robot::RobotControl_getSpeed_Result &getSpeed()
                  { return _pd_getSpeed; }
      void getSpeed( const  class robot::RobotControl_getSpeed_Result &  _v) {
        clear();
        _pd_getSpeed = _v;
        this->discriminator(robot::RobotControl_getSpeed_hash);
      }
      const  class robot::RobotControl_getStatus_Result &getStatus() const
                  { return _pd_getStatus; }
       class robot::RobotControl_getStatus_Result &getStatus()
                  { return _pd_getStatus; }
      void getStatus( const  class robot::RobotControl_getStatus_Result &  _v) {
        clear();
        _pd_getStatus = _v;
        this->discriminator(robot::RobotControl_getStatus_hash);
      }
      DDS::rpc::UnknownOperation unknownOp() const { return _u._pd_unknownOp; }
      void unknownOp(  DDS::rpc::UnknownOperation _v) {
        clear();
        _u._pd_unknownOp = _v;
        this->discriminator(0);
      }
  };
```

Figure 16: Example RobotControl_Return

```
  struct COREDX_TS_STRUCT_EXPORT RobotControl_Request {
    public:
      /** Constructor, Copy Constructor, Destructor, Assignment operator */
      RobotControl_Request();
      ~RobotControl_Request();
      RobotControl_Request( const RobotControl_Request & other );
      RobotControl_Request& operator=( const RobotControl_Request & other);

      void init();
      void clear();
      void copy( const robot::RobotControl_Request * instance );

      int  get_marshal_size(int offset, int just_keys) const ;
      int  marshal_cdr(unsigned char * buf, int offset, int stream_len,
```

```
                unsigned char swap, int just_keys) const ;
      int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
      int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len,
                unsigned char swap, int just_keys);
      int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

      /* Member vars*/
      DDS::SampleIdentity_t * request_id() { return &this->header.requestId; }
      struct DDS::rpc::RequestHeader  header;
      class robot::RobotControl_Call  data;

      typedef RobotControl_RequestTypeSupport TypeSupport;
      typedef RobotControl_RequestDataReader DataReader;
      typedef RobotControl_RequestDataWriter DataWriter;
      typedef RobotControl_RequestPtrSeq Seq;

    private:

  }; //robot::RobotControl_Request
```

Figure 17: Example RobotControl_Request

```
  struct COREDX_TS_STRUCT_EXPORT RobotControl_Reply {
    public:
      /** Constructor, Copy Constructor, Destructor, Assignment operator */
      RobotControl_Reply();
      ~RobotControl_Reply();
      RobotControl_Reply( const RobotControl_Reply & other );
      RobotControl_Reply& operator=( const RobotControl_Reply & other);

      void init();
      void clear();
      void copy( const robot::RobotControl_Reply * instance );

      int  get_marshal_size(int offset, int just_keys) const ;
      int  marshal_cdr(unsigned char * buf, int offset, int stream_len,
                unsigned char swap, int just_keys) const ;
      int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
      int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len,
                unsigned char swap, int just_keys);
      int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

      /* Member vars*/
      DDS::SampleIdentity_t * related_id() { return &this->header.relatedRequestId; }
      struct DDS::rpc::ReplyHeader  header;
      class robot::RobotControl_Return  data;

      typedef RobotControl_ReplyTypeSupport TypeSupport;
      typedef RobotControl_ReplyDataReader DataReader;
      typedef RobotControl_ReplyDataWriter DataWriter;
      typedef RobotControl_ReplyPtrSeq Seq;

    private:
```

```
  }; //robot::RobotControl_Reply
```

Figure 18: Example RobotControl_Reply

With those data types at hand, the application can use the Request/Reply API to initiate service calls and process returned data.

A server implementation instantiates a Replier with template parameters RobotControl_Request and RobotControl_Reply. It then can set up to process requests and send replies. This can be accomplished in a number of ways. For example, it could simply enter a loop to call Replier::receive_request() and Replier::send_reply().

A snippet of code showing a simple Replier implementation for RobotControl is shown below.

```
DDS::rpc::ReplierParams rep_params;
rep_params.
  domain_participant(dp).
  service_name(service_name).
  instance_name(instance_name);

DDS::rpc::Replier<robot::RobotControl_Request, robot::RobotControl_Reply>
  replier(rep_params);

while (!all_done)
  {
    // RobotControl_Request request;
    DDS::Duration_t delay;
    DDS::Sample<robot::RobotControl_Request> req_sample;

    delay.sec     = 0;
    delay.nanosec = NSEC_PER_SEC / 2;

    if (replier.receive_request(req_sample, delay))
      {
        robot::RobotControl_Request       & req =  req_sample;
        robot::RobotControl_Reply         * reply;
        reply = handle_request(req);
        if (reply)
          {
            DDS::WriteSample<robot::RobotControl_Reply> reply_sample(*reply);
            replier.send_reply(reply_sample,
                  ((robot::RobotControl_Request)req).header.requestId);
            delete reply;
          }
      }
  }

replier.close();
```

Figure 19: Example RobotControl Replier

A client application instantiates a 'Requester' object, with the “_Request” type as a template parameter. This object is capable of writing the appropriate data type to invoke an operation on a Replier object. The following code illustrates creating a client of the RobotControl service.

```
RequesterParams req_params;
req_params.
  domain_participant(participant).
  service_name(“RobotControl”);

Requester<robot::RobotControl_Request, robot::RobotControl_Reply>
  requester(req_params);

requester.wait_for_service(); /* wait for any service to be discovered */

// set up to invoke getStatus()
robot::RobotControl_Request         request;
robot::RobotControl_getStatus_In    getStatus;
DDS::Sample<robot::RobotControl_Reply>   reply;

requester.send_request(request);
if (requester.receive_reply(reply, request.header.requestId))
  ; // HANDLE REPLY
```

Figure 20: Example RobotControl Requester

6 Usage

To use RPC over DDS in an application, it is necessary to include certain header files or packages and to have certain libraries available at run time. Each programming language environment has different requirements that are documented here.

6.1 C Language

[The RPC API for C is not yet available.]

6.2 C++ Language

The C++ RPC Request-Reply API is declared in the “request_reply.hh” header file.

```
#include <dds/request_reply.hh>
```

The C++ RPC Function-Call API is declared in the “function_call.hh” header file.

```
#include <dds/function_call.hh>
```

Because the RPC API implementation uses ContentFilters, it must be linked against the CoreDX DDS libraries that support ContentFilter.]

Operating System	static library	dynamic library	dependencies
linux	libdds_rpc_cpp.a	libdds_rpc_cpp.so	libdds_cpp_cf.a libdds_cf.so
Windows	dds_rpc_cpp_static.lib	dds_rpc_cpp.dll	dds_cpp_cf.lib dds_cf.dll

6.3 C# Language

The C# RPC over DDS API is defined in namespace com.toc.coredx.DDS.rpc.

```
using com.toc.coredx.DDS.rpc;
```

At run-time, the library search path (eg, MONO_PATH, or LD_LIBRARY_PATH), must include a path that contains **dds_rpc_csharp.dll** and its dependency **coredx_csharp.dll**.

6.4 Java Language

The C# RPC over DDS API is defined in namespace com.toc.coredx.DDS.rpc.

```
import com.toc.coredx.DDS.rpc.*;
```

At run-time, the Java class-path must include the RPC and DDS jars: **coredx_rpc.jar coredx_dds.jar**

Also the library search path (eg, LD_LIBRARY_PATH), must include a path that contains the native library **libdds_java.so** (linux) or **dds_java.dll** (windows).

7 Availability

The initial implementation of RPC over DDS supports the C++, C# and Java language mappings. A C-language mapping is under development, and should be available shortly after the initial release.