An OMG® Remote Procedure Call over DDS Publication

Remote Procedure Call over DDS (DDS-RPC)

v1.0

OMG Document Number: formal/2017-04-01

Date: April, 2017

Standard document URL: http://www.omg.org/spec/DDS-RPC/1.0

Normative Machine Consumable File(s):

http://www.omg.org/spec/DDS-RPC/20160215/omg-dds-rpc-cxx.zip

http://www.omg.org/spec/DDS-RPC/20160215/omg-dds-rpc-java.zip

http://www.omg.org/spec/DDS-RPC/20160215/omg-dds-rpc-idl.zip

OMG's Issue Reporting

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

This page intentionally left blank.

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language®); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel™); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

Signal and Image Processing Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text, table text, bullets

Helvetica/Arial – 9 or **10 pt. Bold:** OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier new/Courier – 10 pt. Bold`: Programming Languages

Helvetica/Arial – 10 pt.: Exceptions

`Courier/Courier New – 9`

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://issues.omg.org/issues/create-new-issue*

1 Scope

The Data Distribution Service is widely used for data-centric publish/subscribe communication in real-time distributed systems. Large distributed systems often need more than one style of communication. For instance, data distribution works great for one-to-many dissemination of information. However, certain other styles of communication namely request/reply and remote method invocation are cumbersome to express using the basic building blocks of DDS. Using two or more middleware frameworks is often not practical due to complexity, cost, and maintenance overhead reasons. As a consequence, developing a standard mechanism for request/reply style bidirectional communication on top of DDS is highly desirable for portability and interoperability. Such facility would allow commands to be naturally represented as remote method invocations. This presents a solution to this problem.

This specification defines a Remote Procedure Calls (RPC) framework using the basic building blocks of DDS, such as topics, types, DataReaders, and DataWriters to provide request/reply semantics. It defines distributed services, described using a service interface, which serves as a shareable contract between service provider and a service consumer. It supports synchronous and asynchronous method invocation. Despite its similarity, it is not intended to be a replacement for CORBA.

2 Conformance

This specification defines two conformance points: Basic and Enhanced.

[1] Basic conformance (mandatory)

[2] Enhanced conformance (optional)

The basic conformance point includes support for the Basic service mapping and both the functional and the request-reply language binding styles.

The enhanced conformance point includes the basic conformance and adds support for the Enhanced Service mapping.

The table below summarizes what is included in each of the conformance points.

Conformance point	**Service Mapping**		**Language Binding Style**	
	Basic	**Enhanced**	**Function-call**	**Request/Reply**
Basic	Included		Included	Included
Enhanced	Included	Included	Included	Included

3 Normative References

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of any of these publications do not apply.

- [**DDS**] *Data Distribution Service for Real-Time Systems Specification*, version 1.2 (OMG document formal/2007-01-01).
- [**RTPS**] The Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol (DDSI-RTPS) (OMG document formal/2010-11-01)

- [**DDS-Xtypes**] *Extensible and Dynamic Topic Types for DDS*, version 1.0 Beta 1 (OMG document ptc/2010-05-12
- [**CORBA**] Common Object Request Broker Architecture (CORBA/ZIOP) version 3.3. OMG documents formal/2012-11-12, formal/2012-11-14, and formal/2012-11-16
- [**IDL35**] Interface Definition Language (IDL), version 3.5, OMG document formal/2014-03-01
- [**EBNF**] ISO/IEC 14977 Information Technology – Syntactic Metalanguage – Extended BNF (first edition)
- [**Java-Grammar**] The Java Language specification, Third Edition. Chapter 8 http://docs.oracle.com/javase/specs/jls/se5.0/html/syntax.html
- [**DDS-Cxx-PSM**] ISO/IEC C++ 2003 Language DDS PSM, version 1.0, OMG document formal/2013-11-01
- [**DDS-Java-PSM**] Java 5 Language PSM for DDS, version 1.0, OMG document formal/2013-11-02
- [**DDS4CCM**] DDS for lightweight CCM, version 1.1, OMG document formal/2012-02-01
- [**IDL2Java**] IDL to Java Language Mapping, version 1.3, OMG document formal/2008-01-14
- [**SOA-RM**] Reference Model for Service Oriented Architecture (SOA-RM), v1.0, October 2006

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Service - a Service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consisent with constraints and policies as specified by the service description. [SOA-RM]

Remote Procedure Call – Remote Procedure Call is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space.

5 Symbols and Abbreviated Terms

- DDS – Data-Distribution Service
- GUID – Global Unique Identifier
- RPC – Remote Procedure Call
- RTPS – Real-Time Publish-Subscribe Protocol
- SN – Sequence Number

6 Additional Information

6.1 Acknowledgements

The following companies submitted this specification:

- Real-Time Innovations, Inc.
- eProsima
- PrismTech

The following companies support this specification:

- Real-Time Innovations, Inc.
- eProsima
- PrismTech
- Twin Oaks Computing, Inc.

This page is intentionally left blank.

7 Remote Procedure Call over Data Distribution Service

7.1 Overview

Large distributed systems often require different communication patterns depending upon the problem at hand. For instance, distribution of sensor data is best achieved using unidirectional *one-to-many* pattern whereas sending commands to a specific device or retrieving configuration of a remote service is best done using bidirectional request/reply communication. Using a single middleware that supports multiple communication styles is a very cost-effective way of developing and maintaining large distributed systems. Data Distribution Service (DDS) is a well-known standard for data-centric publish-subscribe communication for real-time distributed systems. DDS excels at providing an abstraction of global data space where applications can publish real-world data and also subscribe to it without temporal or spatial coupling.

DDS, however, is cumbersome to use for bidirectional communication in the sense of request-reply pattern. The pattern can be expressed using the basic building blocks of DDS, however, substantial plumbing must be created manually to achieve the desired effect. As a consequence, it is fair to say that request/reply style communication is not *first-class* in DDS. The intent of this specification is to specify higher-level abstractions built on top of DDS to achieve *first-class* request/reply communication. It is also the intent of this specification to facilitate portability, interoperability, and promote data-centric view for request/reply communication so that the architectural benefits of using DDS can be leveraged in request/reply communication.

7.2 General Concepts

7.2.1 Architecture

Remote Procedure Call necessarily has two participants: a *client* and a *service*. Structurally, every client uses a data writer for sending requests and a data reader for receiving replies. Symmetrically, every *service* uses a data reader for receiving the requests and a data writer for sending the replies.

Figure 7.1 shows the high-level architecture of the remote procedure call over DDS. The client consists of a data writer to publish the sample that represents remote procedure call on the *call* topic. Correspondingly, the service implementation contains a data reader to read the method name and the parameters (i.e., Foo). The service computes the return values (i.e., Bar) to be sent back to the client on the *Return* topic. (The service implementation details are not shown.)

The data reader at the client side receives the response, which is delivered to the application. To ensure that the client receives a response to a previous call made by itself, a content-based filter is used by the reader at the client-side. This ensures that responses for remote invocations of other clients are filtered.

It is possible for a client to have more than one outstanding request, particularly when asynchronous invocations are used. In such cases, it is critical to correlate requests with responses. As a consequence, each individual request must be correlated with the corresponding reply. Requests, like all samples in the DDS data space, are identified using a unique `SampleIdentity` defined as a struct composed of `GUID_t` and a `SequenceNumber_t` defined in sub clause 7.5.1.1.1 Common Types. When a service implementation sends a reply to a specific remote invocation, it is necessary to identify the original request by providing the *sample-identity* of the request. Note that a reply data sample has its own unique message-id (sample identity), which represents the reply message itself and is independent of the request sample-identity.

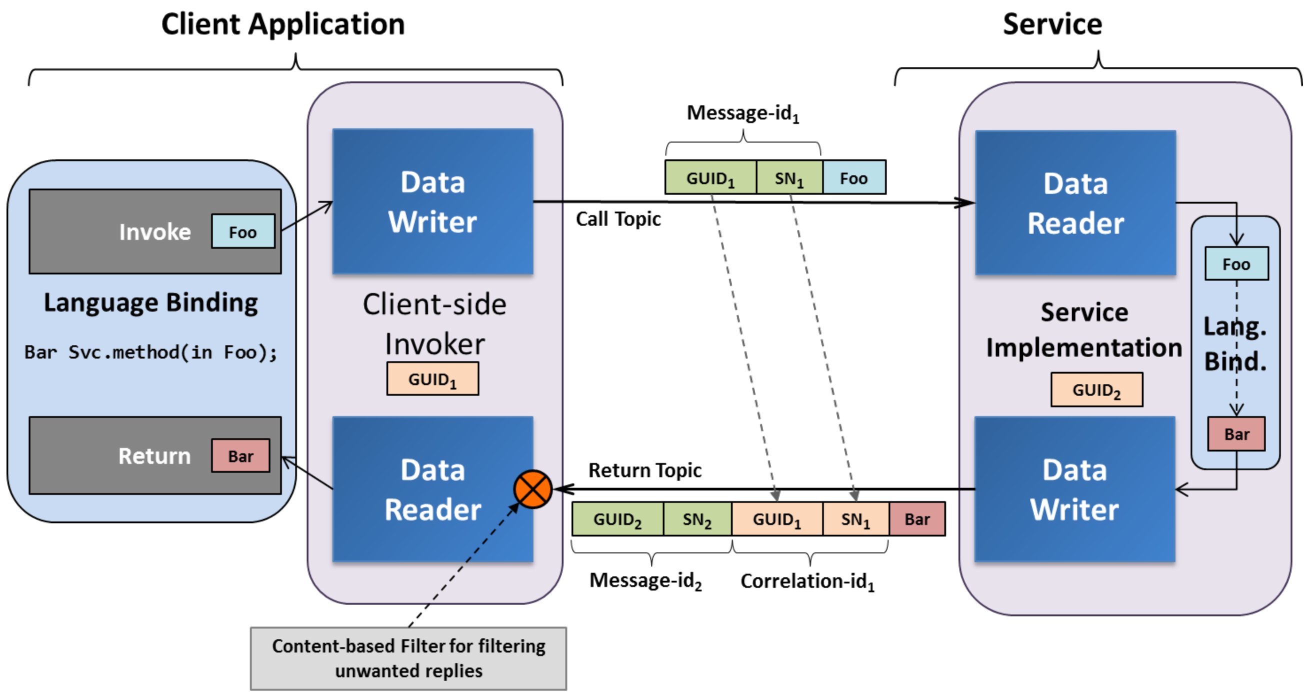

Figure 7.1 - Conceptual View of Remote Procedure Call over DDS

7.2.2 Language Binding Styles for RPC over DDS

Language binding style determines how the client API is exposed to the programmer and how the service implementation receives notification of the arriving requests. This specification includes a higher-level language binding with *function-call* style and a lower-level language binding with *request/reply* style.

7.2.2.1 Function-call Style

The function-call style is conceptually analogous to Java RMI, .NET WFC Service Contracts, or CORBA. To provide function-call style, a common approach is to generate *stubs* that serve as client-side proxies for remote operations and *skeletons* to support service-side implementations. The look-and-feel is like a local function invocation. A code generator generates *stub* and *skeleton* classes from an interface specification. The generated code is used by the client and service implementation. An advantage of such a mapping is that the look and feel of the client-side program and the service implementation is just like a native method call.

7.2.2.2 Request/Reply Style

The request/reply style makes no effort to make the remote invocation look like a function call. Instead, it provides a general-purpose API to send and receive messages. The programmer is responsible for populating the request messages (a.k.a. samples) at the client side and the reply messages on the service side. In that sense it is lower-level language binding compared to the function-call semantics.

The request/reply style provides a flat interface, such as *send_request*, *receive_request*, and *send_reply, receive_reply,* which substantially simplifies language binding as no code generation is necessary beyond the request/reply types. However, remote procedure call does not appear *first-class* to the programmer.

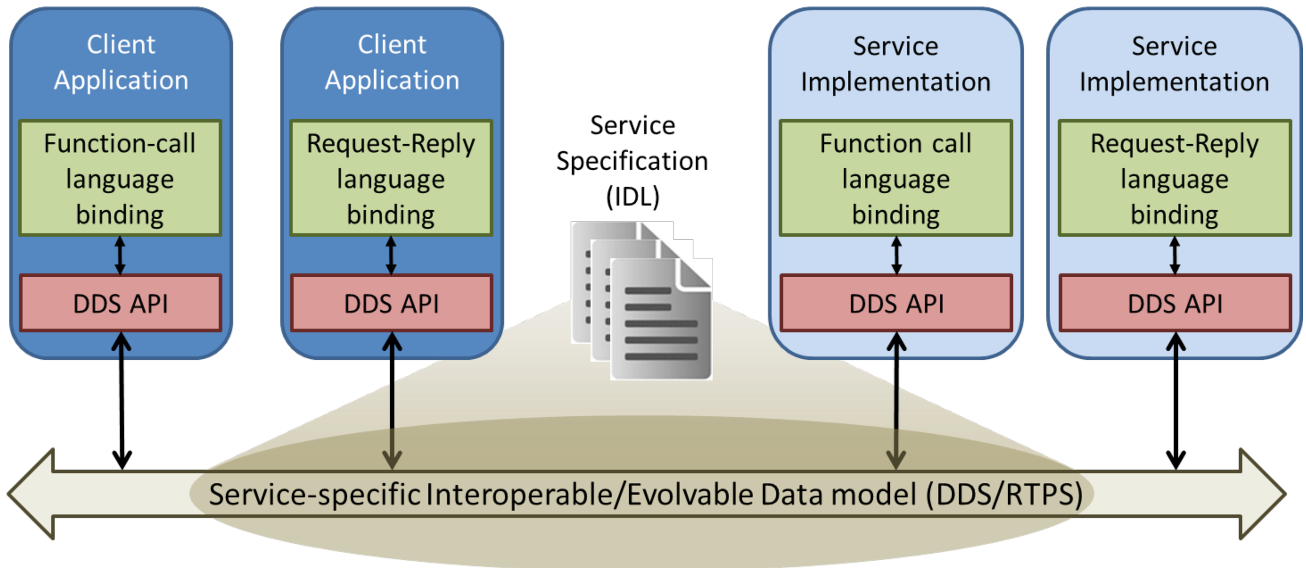

Figure 7.2 - Strong decoupling of language binding (function-call and request /reply) from the service-specific data model

7.2.2.3 Pros and Cons of each Language Binding Style

The function-call style is natural to programmers due to its familiar look-and-feel. Sending of the request message and reception of the corresponding reply is abstracted away using *proxy* objects on the client side. Request-reply style, on the other hand, is more explicit about exchanging messages and therefore can be used to implement complex interactions between the client and the server. For example, the *command-response* pattern typically involves multiple replies to the same request. (e.g., completion percentage status). Request-reply style can easily implement such a pattern without polling. For a given request, a service may simply produce multiple replies with the same request-id as that of the original request. The client correlates the replies with the original request using the request-id. The function-call style must use application-level polling or asynchronous callbacks if multiple replies are expected by the client. This is because the *single-entry-single-return* semantics restrict underlying messaging.

Furthermore, request-reply style is inherently asynchronous because invocation of the service is separated from reception of replies. Multiple replies (for a single request) may be consumed one at a time or in a batch. The API for request-reply style often simplifies code-generation requirements because stubs and skeletons are not required. Finally, the request-reply style is strongly typed and this specification uses templates in case of C++ and generics in case of Java to provide service-specific type-rich language bindings.

It is important to note that the client and service sides are not coupled with respect to the language binding styles. Thanks to the strong separation imposed by the mapping of interfaces to topic types. It is possible for a client to use function-call style language binding to invoke remote operations on a server that uses request/reply style language binding to implement the service. The converse is also true. Furthermore, it is also possible for the stubs and skeletons of the functional style to use the request/reply language binding under the hood.

In light of the above observation, a conforming implementation to this specification shall support both styles of language binding.

7.2.3 Request-Reply Correlation

Request-reply correlation requires an ability to retrieve the sample identity (GUID, SN) at both the requester and service side. The requester needs to know the sample identity because it needs to correlate the reply with the original request. The service implementation also needs to retrieve the sample identity of the request so that it can use it to relate the reply sample to the request that triggered it.

This specification makes important distinction in how the information necessary for correlation is propagated. The sample-identity can be propagated either *implicitly* or *explicitly*. Explicit sample-identity implies that the request-id is visible in the top-level data type for the DDS topic. Implicit sample-identity, on the other hand, implies that the request sample-identity is communicated via extensibility mechanisms supported by the DDS-RTPS protocol. Specifically using the *inlineQoS* sub-message element. See sub clause 8.3.7.2 of the DDS-RTPS specification [RTPS].

In both cases, the specification provides APIs to get the request-id at the client side and get/set the request-id at the service side.

7.2.4 Basic and Enhanced Service Mapping for RPC over DDS

This specification includes two mappings for interfaces to DDS topics and types called “Basic” and “Enhanced.”

- The Basic service mapping enables RPC over DDS functionality without any extensions to the [DDS] and [DDS-RTPS] specifications. It uses explicit request-id for correlation.
- The Enhanced service mapping uses implicit request-id for correlation, allows use of the additional data-types defined in DDS-XTypes, uses DDS-XTypes for type-compatibility checking, and provides more robust service discovery.

The following table summarizes the key aspects of the Basic and Enhanced service mapping profiles.

Mapping Aspect	Basic Service Mapping Profile	Enhanced Service Mapping Profile
Correlation Information (request-id)	Explicitly added to the data-type	Implicit. Correlation Information appears on the Sample meta-data.
Topic Mapping	One request topic and one reply topic per interface. 2*N for a hierarchy of N interfaces.	One request and one reply topic per interface independent of interface hierarchies.
Type Mapping	Synthesized types compatible with DDS 1.3 compliant implementations.	Use facilities of DDS-XTypes for type descriptions, annotations, and type-compatibility checks.
Discovery	No special extensions.	Robust service discovery as described in sub clause 7.6.

7.2.5 Interoperability

Client and service interoperability requires both sides to use the same service mapping. Basic and Enhanced Service Mappings can be mixed in an application but for any given service both client and the service side must use the same service mapping. It is therefore considered part of the service interface contract.

The Basic and Enhanced Service mappings are independent of the language binding style. For instance, it is possible for clients and service implementations to communicate using different language binding styles as long as their service mappings match.

7.3 Service Definition

A service definition is provided using the following two alternatives.

a) **Interface**: An interface is a description of the methods/operations and attributes the service implements. It is a *provided* interface. A service may implement one or more interfaces related by inheritance (single or multiple).

b) **A Pair of Types:** A service specification may simply include a pair of request and reply types. The request and reply types may be the same. The pair of types may not correspond to an interface. However, all service descriptions correspond to a pair of types. In that sense the pair-of-types mechanism of defining a service is strictly more general than interfaces.

This specification uses OMG Interface Definition Language (IDL) or Java 1.5 as a concrete syntax to define services with either of the mechanisms above.

7.3.1 Service Definition in IDL

This specification uses the interface definition syntax specified in [IDL]. Additionally, annotations are supported to provide extra information. Service definition in IDL does not specify whether an operation is synchronous or asynchronous, which is a run-time concern and not intrinsic to the interface contract. Therefore, this specification provides synchronous and asynchronous invocation capabilities only at the language binding level.

Non-normative: The definition of the IDL syntax as part of this specification is a transient situation. Once IDL 4.0 is adopted this specification will be able to simply reference the appropriate IDL syntax building blocks defined in IDL 4.0.

7.3.1.1 Service Definition in IDL for the Basic Service Mapping

The BNF grammar used to define the IDL syntax uses the same production rules as in sub clause 7.4 (IDL Grammar) in the [IDL] specification and sub clause 7.3.1.12.1 (New Productions) of the [DDS-XTypes] specification.

The IDL 3.5 grammar shall be modified with the productions shown below. These productions are numbered using the same numbers as in the IDL 3.5 document.

Note, [ ] represents optional.

```
(1)  <specification>   ::=  <definition>+

(2)  <definition>     ::=    <type_dcl> ";" <ann_appl_post>
                           | <const_dcl> ";"
                           | <except_dcl> ";"
                           | <interface> ";"
                           | <module> ";"
                           | <value> ";"
                           | <annotation> ";" <ann_appl_post>

(7) <interface_header> ::= [ <annotation> ] "interface" <identifier>
                             [ <interface_inheritance_spec> ]
```

(9) <export> ::= <attr_dcl> “;”
| <op_dcl> “;”

(42) <type_dcl> ::= [<annotation>] “typedef” <type_declarator>
| [<annotation>] <struct_type>
| <union_type>
| <enum_type>
| “native” <simple_declarator>
| <constr_forward_decl>

(87) <op_dcl> ::= [<annotation>] <op_type_spec> <identifier> <parameter_dcls>
[<raises_expr>]

(91) <param_dcl> ::= [<annotation>] <param_attribute> <param_type_spec>
<simple_declarator>

(104) <readonly_attr_spec> ::= [<annotation>] “readonly” “attribute”
<param_type_spec> <readonly_attr_declarator>

(106) <attr_spec> ::= [<annotation>] “attribute” <param_type_spec>
<attr_declarator>

The <annotations> production used above is defined in sub clause 7.3.1.12.1 of the [DDS-XTypes] specification.

Design Rationale (non-normative)

The table below provides the justification for the modified production rules.

Reference	Origin	Explanation
(1)	IDL 3.5	This is the root production rule. Modified to remove the CORBA-specific import statement.
(2)	IDL 3.5	Modified to remove the productions for related to CORBA Repository Identity and CCM. Specifically <type_id_dcl>, <type_prefix_dcl>, <event>, <component>, and <home_dcl>. The modified rule also adds support for annotation declarations. The <annotation> and <ann_appl_post> productions are defined in sub clause 7.3.1.12.1 of the [DDS-XTypes] specification.
(7)	IDL 3.5	Modified to add support for interface annotations. The <annotation> production is defined in sub clause 7.3.1.12.1 of the [DDS-XTypes] specification.

(9)	IDL 3.5	Modified to remove productions related to CORBA Repository Identity. Specifically <type_id_dcl> and <type_prefix_dcl> This production also removes the rules that allow embedding declarations of types, constants, and exceptions within an interface. Specifically the <type_dcl>, | <const_dcl>, and | <except_dcl>.
(42)	IDL 3.5	Added <annotation> production at the beginning of struct and typedef declarators. This allows a structured type to be a request or reply type.
(87)	IDL 3.5	Modified to add support for annotations on operations. Removed the CORBA-specific "oneway" modifier and the "context" expressions.
(91)	IDL 3.5	Modified to add support for annotations on the operation parameters.
(104)	IDL 3.5	Modified to add support for annotations on read-only attributes.
(106)	IDL 3.5	Modified to add support for annotations on attributes.
DDS-XTypes	[DDS-XTypes] sub clause 7.3.1.12.2	These productions add support for annotating types and the new IDL types defined in the [DDS-XTypes] specification.

The use of the <annotations> production from DDS-XTypes does not mean that the underlying DDS implementation needs to support DDS-XTypes. The Basic Service Mapping uses annotations only for interface declarations and not on regular type declarations. These interface annotations are resolved in the mapping such that the resulting IDL used by DDS does not have annotations.

7.3.1.2 Service Definition in IDL for the Enhanced Service Mapping

The Enhanced Service Mapping allows use of the full type-system defined in the DDS-XTypes specification in the declaration of interface attributes, operation parameters, and return values.

The Enhanced Service Mapping extends the IDL productions defined in the Basic Service Mapping with all the productions (i.e., not just related to annotations) in sub clause 7.3.1.12.1 (New Productions) of the DDS-XTypes specification.

In addition, the Enhanced Service Mapping also uses all the modified productions defined in sub clause 7.3.1.12.2 (Modified Productions) of the [DDS-Xtypes] specification with the exception of the production for <definition>, which shall remain, as defined for the Basic Service Mapping.

7.3.1.3 Example of an Interface in IDL (Non-normative)

```
module robot {
exception TooFast {};
enum Command { START_COMMAND, STOP_COMMAND };
struct Status {
 string msg;
};
@DDSService
```

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

7.3.1.4 Service Definition in IDL Using a Pair of Types

A service definition may simply include a pair of types, which may be the same. The request and reply types shall be marked as such using the `@RPCRequestType` and `@RPCReplyType` annotations. Only `struct` types shall be marked as request/reply types.

In the Basic Service Mapping, the types annotated as `@RPCRequestType` shall have a member named `header` of type `dds::rpc::RequestHeader` and the types annotated as `@RPCReplyType` shall have a member named `header` of type `dds::rpc::RequestHeader`. See sub clause 7.5.1.1.1 Common Types for the request and reply header types.

When using the Enhanced Service Mapping, the annotations are not necessary and the special data members must not be defined.

7.3.2 Service Definition in Java

The BNF grammar used to define the Java syntax used in [Java-Grammar]. Conforming implementations shall support the syntax generated by the following modified productions from the Java grammar.

Note, [ ] represents optional and { } represents zero or more occurrences.

```
InterfaceDeclaration:
      NormalInterfaceDeclaration
NormalInterfaceDeclaration:
     interface Identifier [extends TypeList] InterfaceBody
InterfaceBody:
        { InterfaceBodyDeclaration {InterfaceBodyDeclaration} }
InterfaceMemberDecl:
        InterfaceMethodOrFieldDecl
        void Identifier VoidInterfaceMethodDeclaratorRest
        InterfaceDeclaration
        ClassDeclaration
NormalClassDeclaration:
     class Identifier [extends Type] [implements TypeList] ClassBody
ClassBody:
```

```
        { ClassBodyDeclaration {ClassBodyDeclaration} }
ClassBodyDeclaration:
        ;
        {Modifier} MemberDecl
MemberDecl:
        MethodOrFieldDecl
MethodOrFieldRest:
        VariableDeclaratorRest
```

Design Rationale (non-normative)

The table below provides the justification for the modified production rules.

Name	Origin	Explanation
`InterfaceDeclaration`	Java 1.5	Removed annotation type declaration.
`NormalInterfaceDeclaration`	Java 1.5	Removed generic type parameters.
`InterfaceBody`	Java 1.5	At least one method must be present in an interface. Note, Java allows empty (marker) interfaces.
`InterfaceMemberDecl`	Java 1.5	Removed generic method declaration.
`NormalClassDeclaration`	Java 1.5	Removed generic type parameters.
`ClassBody`	Java 1.5	At least one data member must be defined in the class body. Note, Java allows empty classes.
`ClassBodyDeclaration`	Java 1.5	Removed Java's static block support.
`MemberDecl`	Java 1.5	Removed method (generic or otherwise) declarator, constructor declarator, nested interface and class declarators. This specification shall support data member declarations inside a Java class.
`MethodOrFieldRest`	Java 1.5	Removed method declarator. This specification shall support data member declarations inside a Java class.

In, *out*, and *inout* parameters are specified using the `@in`, `@out` and `@inout` annotations. By default the parameters are read only, and the use of `@in` is optional. One or more operations may be marked oneway using the `@oneway` annotation.

7.3.2.1 Example of an interface in Java (Non-normative)

```
Public class  TooFast extends Exception
{
}
@DDSService
public interface RobotControl
{
 void  command(Command com);
 float setSpeed(float speed) throws (TooFast);
 float getSpeed();
 void  getStatus(out Status status);
};
```

7.3.2.2 Service Definition in Java Using a Pair of Types

A service definition may simply include a pair of types, which may be the same. The request and reply types shall be marked as such using the `@RPCRequestType` and `@RPCReplyType` annotations. Only `class` types shall be marked as request/reply types.

In the Basic Service Mapping, the types annotated as `@RPCRequestType` shall have a member named `header` of type `dds.rpc.RequestHeader` and the types annotated as `@RPCReplyType` shall have a member named `header` of type `dds.rpc.RequestHeader`. See sub clause 7.5.1.1.1 Common Types for the request and reply header types.

When using the Enhanced Service Mapping, the annotations are not necessary and the special data members must not be defined.

7.4 Mapping Service Specification to DDS Topics

7.4.1 Rules for Synthesizing DDS Topic Names

Request and reply topic names use the following BNF grammar.

<topic_name> ::= <interface_name> "_" <service_name> "_" ["Request" | "Reply"]
| <user_def_alpha_num>
<service_name> ::= "Service"
| <user_def_alpha_num>
<user_def_alpha_num> ::= `^[[:alnum:]_ ]+$`

Topic name is either a user-defined string literal or a composite string consisting **<interface_name>** and **<service_name>.** The **<interface_name>** non-terminal represents the fully-qualified name of the interface, which is captured automatically for interface-based service definitions. A fully qualified interface-name includes concatenation of module names separates by underscores followed by the name of the interface. The **<service_name>** non-terminal may be user-supplied. When it is not, it defaults to “Service.” If the **<service_name>** is specified by the user, “**Request”** and “**Reply”** topic suffixes are used automatically. When **<service_name>** and user-defined topic names are both provided by the user, the user-defined topic names take precedence.

For the request-reply style language binding, **<interface_name>** and the following underscore shall not be captured automatically.

Note that the **<user_def_alpha_num>** non-terminal is a regular expression that includes alphanumeric characters (a-zA-Z0-9), underscore and space.

7.4.2 Basic Service Mapping

The Basic Service mapping maps every interface to a request topic and a reply topic. It provides three alternative mechanisms to specify the names of the topics. It is possible to use different mechanism at the client and server sides. However, to ensure successful end-point matching, the topic names must match.

In case of an interface inheritance hierarchy, including multiple inheritance, each interface in the hierarchy shall be mapped to its own pair of request and reply topics.

7.4.2.1 Default Topic Names

The default topic names are synthesized using the rules defined in sub clause 7.4.1 Rules for Synthesizing DDS Topic Names.

7.4.2.2 Specifying Topic Names using Annotations

Interfaces and the request/reply types may be annotated to specify the names of the request and reply topics. `@DDSRequestTopic` and `@DDSReplyTopic` annotations are pre-defined for this purpose. They are defined using the [DDS-XTypes] notation as follows:

```
module dds {
module rpc {

@Annotation
local interface DDSRequestTopic {
 attribute String name;
};



 @Annotation
local interface DDSReplyTopic {
 attribute String name;
};
};
};
```

Note that support for [DDS-XTypes] in the underlying DDS implementation is not required to interpret the annotations. These annotations simply control the generated DDS wrapper code.

Non-Normative Example: The `RobotControl` interface may use one or both of the annotations shown below.

```
@DDSService
@DDSRequestTopic (name="RobotRequestTopic")
@DDSReplyTopic (name="RobotReplyTopic")

interface RobotControl
{
    void  command(Command.com);
    ...
}
```

7.4.2.3 Specifying Topic Names at Run-time

DDS Topic names may also be specified at run-time. The `ServiceParams`, `ClientParams`, `RequesterParams`, and `ReplierParams` classes (7.11.1.4 Summary of C++ Request-Reply Style Language Binding) provide functions to specify the service name and the topic name suffix. When used as such, the rules defined in 7.4.1 Rules for Synthesizing DDS Topic Names apply.

When more than one method of specifying topic names is used, the run-time specification shall take precedence over the IDL annotations and the default mechanism in that order.

7.4.3 The Enhanced Service Mapping

To support single and multiple inheritance of interfaces, the Enhanced Service mapping introduces a notion called "topic aliases." A topic-alias is an alternative name for a topic. A topic may have one or more topic aliases. A topic-alias does not introduce a new topic. It simply indicates the fact that a given topic may be known by different names. Topic aliases are announced during the endpoint discovery protocol using either of the `PublicationBuiltinTopicDataExt` and `SubscriptionBuiltinTopicDataExt` structures described in 7.6.2.1 Extensions to the DDS Discovery Builtin Topics.

The Enhanced Service Mapping shall map an interface to exactly one request topic and a reply topic. The names of the topics are obtained using the rules defined in 7.4.1 Rules for Synthesizing DDS Topic Names. In case of single and multiple inheritance, the request and reply topics shall additionally have topic-aliases, which are obtained using the rules in sub clause 7.4.1 for each parent interface.

Non-normative example: The following example shows the topic names and topic-aliases for the `Calculator` hierarchy while using the function-call style language binding. Depending upon whether the user has supplied the service name and/or topic-suffixes, different outcomes are produced.

<table>
<tr><td colspan="4"><code>interface Adder { ... };</code>
<code>interface Subtractor { ... };</code>
<code>interface Calculator : Adder, Subtractor { ... }</code></td></tr>
<tr><th colspan="2"></th><th>Topic name</th><th>Topic aliases</th></tr>
<tr><td rowspan="2">Everything Default</td><td>Request</td><td>Calculator_Service_Request</td><td>Adder_Service_Request,
Subtractor_Service_Request</td></tr>
<tr><td>Reply</td><td>Calculator_Service_Reply</td><td>Adder_Service_Reply,
Subtractor_Service_Reply</td></tr>
<tr><td rowspan="2">User-defined service name</td><td>Request</td><td>Calculator_${servicename}_Request</td><td>Adder_${servicename}_Request,
Subtractor_${servicename}_Request</td></tr>
<tr><td>Reply</td><td>Calculator_${servicename}_Reply</td><td>Adder_${servicename}_Reply,
Subtractor_${servicename}_Reply</td></tr>
<tr><td rowspan="2">User-defined topic suffixes</td><td>Request</td><td>Calculator_${topic-suffix-request}</td><td>Adder_${topic-suffix-request},
Subtractor_${topic-suffix-request}</td></tr>
<tr><td>Reply</td><td>Calculator_${topic-suffix-reply}</td><td>Adder_${topic-suffix-reply},
Subtractor_${topic-suffix-reply}</td></tr>
</table>

For the request-reply style language binding, the interfaces name shall not be annexed automatically.

7.4.3.1 Default Topic Names

The default topic names are synthesized as described above.

7.4.3.2 Specifying Topic Names Using Annotations

Interfaces and the request/reply types may be annotated to specify the names of the request and reply topics. The same built-in `@DDSRequestTopic` and `@DDSReplyTopic` annotations defined in the Basic Service Mapping may be used for this purpose.

7.4.3.3 Specifying Topic Names at Run-time

DDS Topic names may also be specified at run-time. The `ServiceParams`, `ClientParams`, `RequesterParams`, and `ReplierParams` classes (7.11.1.4 Summary of C++ Request-Reply Style Language Binding) provide functions to specify the service name and the topic name suffix. When used as such, the rules defined in 7.4.1 Rules for Synthesizing DDS Topic Names apply.

When more than one method of specifying topic names is used, the run-time specification shall take precedence over the IDL annotations and the default mechanism in that order.

7.5 Mapping Service Specification to DDS Topics Types

The *request* and *reply* DDS Topic types shall be synthesized from the interface definition.

7.5.1 Interface Mapping

7.5.1.1 Basic Service Mapping of Interfaces

The Basic Service Mapping maps interfaces to data-types that can be used by DDS version 1.3 compliant implementations, which may lack support for the [DDS-XTypes] specification.

7.5.1.1.1 Common Types

All the generated types as per the Basic Service Mapping use a set of common types. Types `EntityId_t`, `GUID_t`, and `SequenceNumber_t` are defined in the [RTPS] specification.

```
module dds {
  typedef octet GuidPrefix_t[12];

  struct EntityId_t   {
    octet entityKey[3];
    octet entityKind;
  };

  struct GUID_t  {
    GuidPrefix_t guidPrefix;
    EntityId_t entityId;
  };

  struct SequenceNumber_t   {
    long high;
    unsigned long low;
  };

  struct SampleIdentity  {
    GUID_t           writer_guid;
    SequenceNumber_t sequence_number;
  };

module rpc {
```

```
typedef octet UnknownOperation;
typedef octet UnknownException;
typedef octet UnusedMember;

enum RemoteExceptionCode_t
{
  REMOTE_EX_OK,
  REMOTE_EX_UNSUPPORTED,
  REMOTE_EX_INVALID_ARGUMENT,
  REMOTE_EX_OUT_OF_RESOURCES,
  REMOTE_EX_UNKNOWN_OPERATION,
  REMOTE_EX_UNKNOWN_EXCEPTION
};

typedef string<255> InstanceName;
struct RequestHeader {
 SampleIndentity_t requestId;
 InstanceName instanceName;
};
struct ReplyHeader {
 dds::SampleIdentity relatedRequestId;
 dds::rpc::RemoteExceptionCode_t remoteEx;
};
} // module rpc
} // module dds
```

7.5.1.1.2 The Hashing Algorithm

In the implied IDL definition in this specification use a `HASH` function to compute a 32-bit hash of strings. The `HASH` function shall use the following pseudo-implementation.

```
long HASH (string arg)
{
  octet md5_hash [16] ;
  md5_hash = compute_md5 (  arg ) ;

return                  md5_hash[0] +
                  256*  md5_hash[1] +

             256*256*   md5_hash[2] +
```

```
        256*256*256*     md5_hash[3];
}
```

Conforming implementation are not required to detect collisions of identifier names of the form *_Hash as in general they are not detectable because IDL modules can be reopened and a user could add a colliding identifier much later a service interface is defined. Therefore, this specification does not require detection of collision of const identifier names for hashes.

7.5.1.1.3 Mapping of Attributes to Implied IDL

Every attribute in the interface maps to *implied* IDL, using the following rules:

1. Each attribute in an interface maps to a pair of IDL operations. get_attribute_<attribute-name> and set_attribute_<attribute-name> in the same interface. It is illegal to have an interface with an attribute named <attribute-name> and user-defined operations named get_attribute_<attribute-name> and set_attribute_<attribute-name>.
2. The return type of the get_attribute_<attribute-name> operation is the same as the attribute's type. This operation accepts no arguments.
3. The return type of the set_attribute_<attribute-name> operation is void and it accepts an argument of the same type as that of the attribute and the name of the argument is the same as the attribute name.
4. Exception types listed in getraises, if any, are treated as if the get_attribute_<attribute-name> operation has the same set of exceptions listed as raises.
5. Exception types listed in setraises, if any, are treated as if the set_attribute_<attribute-name> operation has the same set of exceptions listed as raises.

7.5.1.1.4 Mapping of Operations to the Request Topic Types

The mapping of an interface operation to a request type is defined according to the following rules. In these rules the token ${interfaceName} shall be replaced with the name of the interface and the token ${operationName}_shall be replaced with the name of the operation. The substituted names are not fully qualified (i.e., any module prefixes shall be removed).

1. Each operation in the interface shall map to an *In* structure with name "${interfaceName}_${operationName}_In."
2. The *In* structure shall be defined within the same module as the original interface.
3. The *In* structure shall contain as members the *in* and *inout* parameters defined in the operation signature.
   a. The members shall appear in the same order as the parameters in the operation, starting from the left.
   b. The names of the members shall be the same as the formal parameter names.
   c. If an operation has no *in* and *inout* parameters the resulting structure shall contain a single member of type UnusedMember and name "dummy."

Non-normative Example: The operations in the RobotControl interface defined in sub clause 7.3.1.3 shall map to the following *In* structures.

```
module robot {

struct RobotControl_command_In {
   Command com;
```

```
};

struct RobotControl_setSpeed_In {
   float speed;
};

struct RobotControl_getSpeed_In {
   dds::rpc::UnusedMember dummy;
};

struct RobotControl_getStatus_In {
   dds::rpc::UnusedMember dummy;
};
}
```

7.5.1.1.5 Mapping of Operations to the Reply Topic Types

Each operation in the interface shall map to an *Out* structure and a *Result* union.The following rules define the *Out* structure.

1. The name of the *Out* structure shall be "${interfaceName}_${operationName}_Out."
2. The *Out* structure shall be defined within the same module as the original interface.
3. The *Out* structure shall contain as members the *out* and *inout* parameters defined in the operation signature. In addition it may contain a member named "return_."
   a. The *Out* structure shall include one member for each *out* and *inout* parameter in the operation signature.
      i. The members shall appear in the same order as the parameters in the operation, starting from the left.
      ii. The name of the members shall be the same as the formal parameter names.
   b. If the operation defines a non-void return value the *Out* structure shall have its last member named "return_." The type of the "return_" member shall be the return type of the operation. In the case where the function does not define a return value this member shall not be present. If an operation has an out argument named `return_` and the operation has a regular return value, the regular return value in the *Out* structure shall be represented as `return_N` member of the appropriate type where N is the first integer in the range 1 to 2^31 that avoids the collision.
   c. If the operation has no return value, no *out/inout* parameters, the *Out* structure shall contain a single member named "dummy" of type `dds::rpc::UnusedMember`.

Non-normative Example: The operations defined in the `RobotControl` interface defined in 7.3.1.3 Example of an Interface in IDL (Non-normative) map to the following *Out* structures.

```
module Robot {
struct RobotControl_command_Out {
   dds::rpc::UnusedMember dummy;
};
```

```
struct RobotControl_setSpeed_Out {
  float return_;
};


struct RobotControl_getSpeed_Out {
  float return_;
};


struct RobotControl_getStatus_Out {
  Status status;
};
}
```

The following rules define the *Result* union:

1. The *Result* union name shall be "${interfaceName}}_${operationName}_Result."
2. The *Result* union discriminator type shall be `long`.
3. The *Result* union shall have a case with label `dds::RETCODE_OK` which is used to represent a successful return.
    a. This case label shall contain a single member with name "result" and type ${interfaceName}_${operationName}_Out.
4. For each exception type raised by the operation,
    a. A constant of type `long` and with name "${exceptionType}_Ex_Hash" shall be available in the same namespace as the interface is defined in.
    b. The value of the constant shall be the `HASH` of the fully qualified name of the exception type where module separator to be used is "::" (2 colons).
5. The union shall have a case label for each exception declared as a possible outcome of the operation.
    a. The integral value of the case label shall be ${exceptionType}_Ex_Hash.
    b. The case label shall contain a single member with name synthetized as the lower-case of the exception name with the suffix "_ex" added.
    c. The type associated with the case member shall be the exception type.

Non-normative Example: The operations defined in the `RobotControl` interface defined in 7.3.1.3 Example of an Interface in IDL (Non-normative) map to the following *Result* unions.

```
module robot {

const long TooFast_Ex_Hash = HASH ("TooFast");

union RobotControl_command_Result switch(long)

{

      case dds::RETCODE_OK:

            RobotControl_Command_Out result;

};


union RobotControl_setSpeed_Result switch(long)
```

```
{
      case dds::RETCODE_OK:
            RobotControl_setSpeed_Out result;
      case TooFast_Ex_Hash:
            TooFast toofast_ex;
};

union RobotControl_getSpeed_Result switch(long)
{
      case dds::RETCODE_OK:
            RobotControl_getSpeed_Out result;
};

union RobotControl_getStatus_Result switch(long)
{
      case dds::RETCODE_OK:
            RobotControl_getStatus_Out result;
};
};// module robot
```

7.5.1.1.6 Mapping of Interfaces to the Request Topic Types

Each interface shall map to a *Call* union and a *Request* structure. The following rules define the *Call* union:

1. The *Call* union name shall be "${interfaceName}_Call" in the same module as the interface.
2. The *Call* union discriminator type shall be of type long.
3. The *Call* union shall have a default case label containing a member named "`unknownOp`" of type `dds::rpc::UnknownOperation`. [**Non-normative design rationale:** Due to interface evolution a client that uses new interface may end up calling a service that implements a previous version of the interface. In that case, the discriminator value will not match any of the existing cases and hence will default to unknown, which should be recognized by the service implementation.]
4. For each operation in the interface, an integral constant of type `long` and name "$(interfaceName}_${operationName}_Hash" shall be present in the same module as the interface. The value of this constant is the HASH of unqualified ${operationName}.
5. The *Call* union shall have a case label for each operation in the interface.
    a. The integral value of the case label shall be the HASH of ${operationName}.
    b. The member name for the case label shall be the operation name.
    c. The type for the case label member shall be ${interfaceName}_${operationName}_In as defined in sub clause 7.5.1.1.2 The Hashing Algorithm.

Non-normative Example: The `RobotControl` interface defined in 7.3.1.3 Example of an Interface in IDL (Non-normative) shall map to the following union `RobotControl_Call`.

```
module robot {
const long RobotControl_command_Hash   = HASH("command");
const long RobotControl_setSpeed_Hash  = HASH("setSpeed");
const long RobotControl_getSpeed_Hash  = HASH("getSpeed");
const long RobotControl_getStatus_Hash = HASH("getStatus");

union RobotControl_Call switch(long)
{
      default
            dds::rpc::UnknownOperation unknownOp;

      case RobotControl_command_Hash:
            RobotControl_command_In command;

      case RobotControl_setSpeed_Hash:
            RobotControl_setSpeed_In setSpeed;

      case RobotControl_getSpeed_Hash:
            RobotControl_getSpeed_In getSpeed;

      case RobotControl_getStatus_Hash:
            RobotControl_getStatus_In getStatus;
};
}
```

The following rules define the *Request* structure:

1. The name of *Request* structure shall be "${interfaceName}_*Request*" in the same module as the interface.
2. The *Request* structure shall have two members:
   a. The first member of the *Request* structure shall be named "header" and be of type `dds::rpc::RequestHeader`.
   b. The second member of the *Request* structure shall be named "data" and be of type ${interfaceName}_Call.

Non-normative Example: The `RobotControl` interface defined in 7.3.1.3 Example of an Interface in IDL (Non-normative) shall map to `RobotControl_Request` defined below.

```
Module robot {
struct RobotControl_Request {
       dds::rpc::RequestHeader header;
       RobotControl_Call       data;
};
}
```

7.5.1.1.7 Mapping of Interfaces to the Reply Topic Types

Each interface shall map to a *Return* union and a *Reply* structure.

The following rules define the *Return* union:

1. The *Return* union name shall be "${interfaceName}_Return" in the same module as the interface.
2. The *Return* union discriminator type shall be of type `long`.
3. The *Return* union shall have a default case label containing a member named "unknownOp" of type `dds::rpc::UnknownOperation`. **[Non-normative rationale:** Unknown operation errors should not be reported back using this member. The `ReplyHeader.remoteEx` member should be used for that. The only reason the `unknownOp` member is used because it is a good practice to define a default for a union.]
4. For each operation,
   a. A constant of type `long` and with name "${interfaceName}_${operationName}_Hash" shall be available in the same namespace as the interface is defined in.
   b. The value of the constant shall be the `HASH` of the name of the operation (not qualified).
5. The *Return* union shall have a case label for each operation in the interface:
   a. The integral value of the case label shall be ${interfaceName}_${operationName}_Hash as computed using the HASH algorithm of the unqualified name of the operation.
   b. The member name for the case label shall be the operation name and the type shall be ${interfaceName}_${operationName}_Result.

Non-normative Example: The `RobotControl` interface defined in 7.3.1.3 Example of an Interface in IDL (Non-normative) shall map to the union `RobotControl_Return` defined below:

```
union RobotControl_Return switch (long)
{
       default:
              dds::rpc::UnknownOperation unknownOp;

       case RobotControl_command_Hash:
              RobotControl_command_Result command;
```

```
        case RobotControl_setSpeed_Hash:
            RobotControl_setSpeed_Result setSpeed;

        case RobotControl_getSpeed_Hash:
            RobotControl_getSpeed_Result getSpeed;

        case RobotControl_getStatus_Hash:
            RobotControl_getStatus_Result getStatus;
};
```

The following rules define the *Reply* structure:

1. The *Reply* structure name shall be "${interfaceName}_*Reply*,".
2. The *Reply* structure shall be defined within the same module as the original interface.
3. The *Reply* structure shall have two members:
    a. The first member of the *Reply* structure shall be named "header" and be of type `dds::rpc::ReplyHeader`.
    b. The second member of the *Reply* type shall be named "data" and be of type ${interfaceName}_Return.

Non-normative Example: The `RobotControl` interface defined in 7.3.1.3 Example of an Interface in IDL (Non-normative) shall map to the structure `RobotControl_Reply` defined below:

```
struct RobotControl_Reply {
        dds::rpc::ReplyHeader header;
        RobotControl_Return   reply;
};
```

7.5.1.1.8 Mapping of Inherited Interfaces to the Request and Reply Topic Types

Inheritance has no effect on the generated structures and unions. A DDS service that implements a derived interface uses two topics for every interface in the hierarchy. As a result, a service implementing a hierarchy of N interfaces, shall have N request topics and N reply topics. Consequently, it will necessitate N DataWriters and N DataReaders. The types and the topic names are obtained as specified in 7.4.2 Basic Service Mapping. The *Request* and *Reply* types for a derived interface includes operations defined only in the derived interface.

Non-normative Example: The Basic Service Mapping for the Calculator interface is shown below.

```
interface Adder {
        long add(long a, long b);
};
interface Subtractor {
```

```
    long sub(long a, long b);
};
interface Calculator : Adder, Subtractor {
    void on();
    void off();
};

struct Adder_add_In {
    long a;
    long b;
};

struct Adder_add_Out {
    long return_;
};
union Adder_add_Result switch (long) {
    default:
        dds::rpc::UnknownException unknownEx;

    case dds::rpc::REMOTE_EX_OK:
        Adder_add_Out result;
};
const long Adder_add_Hash = HASH("add");

union Adder_Call switch (long) {
    default:
        dds::rpc::UnknownOperation unknownOp;

    case Adder_add_Hash:
        Adder_add_In add;
};

struct Adder_Request {
    dds::rpc::RequestHeader header;
    Adder_Call data;
};
```

```
union Adder_Return switch (long) {
      default:
            UnknownOperation unknownOp;

case Adder_add_Hash:
      Adder_add_Result add;
};

struct Adder_Reply {
      dds::rpc::ReplyHeader header;
      Adder_Return data;
};

struct Subtractor_sub_In {
      long a;
      long b;
};

struct Subtractor_sub_Out {
      long return_;
};
union Subtractor_sub_Result switch (long) {
      default:
            dds::rpc::UnknownException unknownEx;
      case dds::rpc::REMOTE_EX_OK:
            Subtractor_sub_Out result;
};
const long Subtractor_sub_Hash = HASH ("sub") ;

union Subtractor_Call switch (long) {
      default:
            dds::rpc::UnknownOperation  unknownOp;
```

```
	case Subtractor_sub_Hash:
		Subtractor_sub_In sub;
};
struct Subtractor_Request {
	dds::rpc::RequestHeader header;
	Subtractor_Call data;
};
union Subtractor_Return switch (long) {
	default:
		dds::rpc::UnknownOperation  unknownOp;

	case Subtractor_sub_Hash:
		Subtractor_sub_Result sub;
};

struct Subtractor_Reply {
	dds::rpc::ReplyHeader header;
	Subtractor_Return data;
};

struct Calculator_on_In {
	dds::rpc::UnusedMember dummy;
};

struct Calculator_off_In {
	dds::rpc::UnusedMember dummy;
};

struct Calculator_on_Out {
	dds::rpc::UnusedMember dummy;
};
union Calculator_on_Result switch (long) {
	default:
		dds::rpc::UnknownException unknownEx;
```

```
      case dds::rpc::REMOTE_EX_OK:
            Calculator_on_Out result;
};

struct Calculator_off_Result  switch (long)  {
      default:
            dds::rpc::UnknownException unknownEx;

      case dds::rpc::REMOTE_EX_OK:
            Calculator_off_Out result;
};

const long Calculator_on_Hash = HASH("on");
const long Calculator_off_Hash = HASH("off");

union Calculator_Call switch (long) {
      default:
            dds::rpc::UnknownOperation  unknownOp;

      case Calculator_on_Hash:
            Calculator_on_In on;

      case Calculator_off_Hash:
            Calculator_off_In off;
};
struct Calculator_Request {
      dds::rpc::RequestHeader header;
      Calculator_Call         data;
};

union Calculator_Return switch (long) {
      default:
            dds::rpc::UnknownOperation  unknownOp;
```

```
        case Calculator_on_Hash:
              Calculator_on_Result on;

        case Calculator_off_Hash:
              Calculator_off_Result off;
};

struct Calculator_Reply {
      dds::rpc::RequestHeader header;
      Calculator_Return       data;
};
```

7.5.1.2 Enhanced Service Mapping of Interfaces

To accurately capture the semantics of the method call invocation and return, this specification defines additional built-in annotations. The following annotations are applicable for the Enhanced Service Mapping only. The Enhanced Service Mapping uses the [DDS-XTypes] type system in addition to the annotations defined in 7.5.1.2.1 Annotations for the Enhanced Service Mapping.

7.5.1.2.1 Annotations for the Enhanced Service Mapping

All annotations below are defined in the `dds::rpc` module.

7.5.1.2.1.1 @Choice Annotation

This specification defines an `@Choice` annotation to capture the semantics of a `union` without using a discriminator. Using unions to indicate which operation is being invoked is brittle. Operations in an interface have set semantics and have no ordering constraints. Union, however, enforces strict association with discriminator values, which are too strict for set semantics. Further, use of unions leads to ambiguities in case of multiple inheritance of interfaces.

The `@Choice` annotation is a placeholder annotation defined as follows.

@Annotation **local interface** Choice { };

The `@Choice` annotation is allowed on structures only. When present, the structure shall be interpreted as if the structure had the `@Extensibility(MUTABLE_EXTENSIBILITY)` annotation and all the members of the structure had the `@Optional` annotation. Furthermore, exactly one member shall be present at any given time.

This specification uses the semantics of `@Choice` for the *return* topic type and to differentiate between normal and exceptional return from a remote method call.

7.5.1.2.1.2 @Autoid Annotation

The `@Autoid` annotation shall be allowed on structures and data members. The structures of the topic types synthesized from the interface shall be annotated as `@Autoid`.

The `@Autoid` annotation indicates that the member-id of each member shall be computed using the HASH algorithm (defined in 7.5.1.1.2 The Hashing Algorithm) applied to the name of the member. As IDL does not support overloading, no two members will have the same name. Consequently, the member ids of two members will be different, unless a hash-collision occurs, which shall be detected by the code generator that processes the IDL interface. Using the MD5 hash for the member id also ensures that the topic types synthesized from the interface definitions are not subject to the order of operation declaration or interface inheritance. Note that the order of interface inheritance is irrelevant to the semantics of an interface. Further, it supports interface evolution (including new operations, operation reorder and new base interfaces) without changing the member ids. The collisions of member-ids produced as a result of `@Autoid` annotation must be detected. If a structure annotated as `@Autoid` has a hash collision due to two or more member names map to the same hash-code, the collision shall be reported as an error.

7.5.1.2.1.3 @Empty Annotation

This specification uses empty IDL structures to capture operations that do not accept any parameters. IDL does not support empty structures. @Empty annotation has been introduced to simplify the mapping rules and support operations that take no arguments. Empty structures, however, are used to maintain consistency between the *call* and *return* structures. Furthermore, eliminating empty structures in the synthesized topic types is undesirable because two or more operations may be empty in which case it becomes ambiguous. The `@Empty` annotation allows a structure to be empty. The code synthesized from an empty structure is implementation dependent.

7.5.1.2.1.4 @Mutable Annotation

This annotation shall be allowed on interface operations. It controls the type-mapping for the operation parameters and return value. By default, adding and removing operation parameters would break compatibility with the previous interface. The use of the @Mutable operation changes the type declaration of the synthesized call and return structures so that they are declared with the `@Extensibility(MUTABLE_EXTENSIBILITY)` annotation.

7.5.1.2.2 Mapping of Operations to the Request Topic Types

The mapping for interface operations is the same as defined in 7.5.1.1.2 The Hashing Algorithm with the following additional rules.

If the operation has the `@Mutable` annotation, then the synthesized *In* structure shall have the annotation `@Extensibility(MUTABLE_EXTENSIBILITY)`.

For example, please see sub clause 7.5.1.1.2 The Hashing Algorithm.

7.5.1.2.3 Mapping of Operations to the ReplyTopic Types

Each operation in the interface shall map to an *Out* structure and a *Result* structure. The rules for mapping the *Out* structure are identical to that defined in 7.5.1.1.5 Mapping of Operations to the Reply Topic Types with the following additional rule.

1. If the operation has the `@Mutable` annotation, then the synthesized *Out* structure shall have the annotation `@Extensibility(MUTABLE_EXTENSIBILITY)`.

The mapping rules for the *Result* structure are as follows.

2. The *Result* structure name shall be "${interfaceName}_${operationName}_Result."
3. The *Result* structure shall have the annotation `@Choice`.
4. The *Result* structure shall have the annotation `@Autoid`.
5. The *Result* structure shall have a first member with name "result" and with type ${interfaceName}_${operationName}_Out.
6. The *Result* structure name shall have members for each exception an operation may raise.
   a. The member name shall be synthetized as the lower-case version of the exception name with the suffix "_ex" added.
   b. The type associated with the member shall be the exception type.
7. If the operation has the `@Mutable` annotation, then the synthesized *Result* structure shall have the annotation `@Extensibility(MUTABLE_EXTENSIBILITY)`.

Non-normative example: The operations defined in the `RobotControl` interface defined earlier map to the following *Result* structures.

```
@Choice @Autoid
struct RobotControl_command_Result {
      RobotControl_command_Out result;

};@Choice @Autoid
struct RobotControl_stop_Result {
      RobotControl_getSpeed_Out result;
};
};@Choice @Autoid
struct RobotControl_setSpeed_Result {
      RobotControl_setSpeed_Out result;
      TooFast toofast_ex;
};

};@Choice @Autoid
struct RobotControl_getSpeed_Result {
      RobotControl_getStatus_Out result;
};
```

7.5.1.2.4 Interface Mapping for the Request Topic Types

Each interface shall map to a *Request* structure.

The following rules define the *Request* structure:

1. The *Request* structure name shall be "${interfaceName}_Request."
2. The *Request* structure shall have the `@Choice` annotation.
3. The *Request* structure shall have the `@Autoid` annotation.
4. The *Request* structure shall contain a member for each operation in the interface.
    a. The member name shall be the operation name.
    b. The member type shall be ${interfaceName}_${operationName}_In.

Non-normative example: The `Robotcontrol` interface defined earlier shall result in the `RobotControl_Call` structure defined below.

```
@Choice @Autoid
struct RobotControl_Request {
```

```
        RobotControl_command_In command;
        RobotControl_setSpeed_In setSpeed;
        RobotControl_getSpeed_In getSpeed;
        RobotControl_getStatus_In getStatus;
};
```

7.5.1.2.5 Interface Mapping for the Reply Type

Each interface shall map to a *Reply* structure.

The following rules define the *Reply* structure. In these rules the token ${interfaceName} shall be replaced with the name of the interface:

1. The *Reply* structure name shall be "${interfaceName}_Reply".
2. The *Reply* structure shall have the `@Choice` annotation.
3. The *Reply* structure shall have the `@Autoid` annotation.
4. The *Reply* structure shall contain a member for each operation in the interface.
    a. The member name shall be the operation name.
    b. The member type shall be ${interfaceName}_${operationName}_Result.
5. The *Reply* structure shall contain a member named "remoteEx" of type `dds::rpc::RemoteException_t`. If an interface has an operation named `remoteEx` (case-sensitive), the corresponding *Reply* structure shall contain a member named `remoteEx_N` and type `RemoteException_t`, where N is the first integer in the range 1 to 2^31 that avoids the collision.

Non-normative example: The `RobotService` interface defined earlier maps to the following structure.

```
@Choice @Autoid
struct RobotControl_Reply {
        RobotControl_command_Result command;
        RobotControl_setSpeed_Result setSpeed;
        RobotControl_getSpeed_Result getSpeed;
        RobotControl_getStatus_Result getStatus;
        dds::rpc::RemoteException_t  remoteEx;
};
```

7.5.1.2.6 Mapping of Inherited Interfaces to Request and Reply Topic Types

A derived interface is interpreted as if all the inherited operations are declared in-place in the derived interface. That is, the inheritance structure is *flattened*.

Non-normative Example: The Enhanced Service Mapping rules applied to the `Calculator` interface hierarchy results in the following request and reply Topic types.

```
@Autoid
struct Adder_add_In {
	long a;
	long b;
};

@Autoid
struct Adder_add_Out {
	long return_;
};
@Choice @Autoid
struct Adder_add_Result {
	Adder_add_Out result;
};

@Choice @Autoid
struct Adder_Request {
	Adder_add_Result   add;
};

@Autoid
struct Subtractor_sub_In {
	long a;
	long b;
};

@Autoid
struct Subtractor_sub_Out {
	long return;
};

@Choice @Autoid
struct Subtractor_sub_Result {
```

```
	Subtractor_sub_Out  result;
};

@Choice @Autoid
struct Subtractor_sub_Reply {
	Subtractor_sub_Result  sub;
	dds::rpc::RemoteException_t  remoteEx;
};

@Autoid
struct Calculator_on_In {
	long dummy;
};
@Autoid
struct Calculator_off_In {
	long dummy;
};

@Autoid
struct Calculator_on_Out {
	long dummy;
};

@Choice @Autoid
struct Calculator_on_Result {
	Calculator_on_Out  result;
};

@Autoid
struct Calculator_off_Out {
	long dummy;
};

@Choice @Autoid
```

```
struct Calculator_off_Result {
      Calculator_off_Out  result;
};

@Choice @Autoid
struct Calculator_Request {
      Adder_add_In        add;
      Adder_sub_In        sub;
      Calculator_on_In    on;
      Calculator_off_In   off;
};


@Choice @Autoid
struct Calculator_Reply {
      Adder_add_Result             add;
      Subtractor_sub_Result        sub;
      Calculator_on_Result         on;
      Calculator_off_Result        off;
      dds::rpc::RemoteException_t  remoteEx;
};
```

The Enhanced Service Mapping leverages the assignability rules defined in the [DDS-XTypes] specification to ensure that the topic types remain compatible in case of inheritance. A client using the base interface can invoke an operation in a service that implements a derived interface as long as the same topic names are used.

7.5.2 Mapping of Error Codes

Error codes are (conceptually) classified as *local* and *remote*. The error codes described in the [DDS] PIM are all *local* error codes. This specification further adds *remote* error codes to communicate service-side error conditions back to the client. Remote error codes are represented using the `RemoteExceptionCode_t` IDL enumeration below.

```
module dds { module rpc {
enum RemoteExceptionCode_t
{
  REMOTE_EX_OK,
  REMOTE_EX_UNSUPPORTED,
```

```
  REMOTE_EX_INVALID_ARGUMENT,
  REMOTE_EX_OUT_OF_RESOURCES,
  REMOTE_EX_UNKNOWN_OPERATION,
  REMOTE_EX_UNKNOWN_EXCEPTION
};

} // module rpc
} // module dds
```

The remote exception codes have the following meanings.

Remote Exception Code	Meaning
REMOTE_EX_OK	The request was executed successfully.
REMOTE_EX_UNSUPPORTED	Operation is valid but it is unsupported (a.k.a. not implemented).
REMOTE_EX_INVALID_ARGUMENT	The value of the parameter passed has an illegal value (e.g., two options active in a `@Choice` structure).
REMOTE_EX_OUT_OF_RESOURCES	The remote service ran out of resources while processing the request.
REMOTE_EX_UNKNOWN_OPERATION	The operation called is unknown.
REMOTE_EX_UNKNOWN_EXCEPTION	A generic, unspecified exception was raised by the service implementation.

Both local and remote exception codes map to exceptions in the C++ and Java language bindings defined in 7.11 Language Bindings. More specifically, all the local return codes map to exceptions as defined in the [DDS-Cxx-PSM] and [DDS-Java-PSM] specifications. The remote error codes also map to exceptions (in `dds::rpc` namespace/package) and the details are provided in sub clause 7.11 Language Bindings.

7.6 Discovering and Matching RPC Services

7.6.1 Client and Service Discovery for the Basic Service Mapping

The Basic Service Mapping relies on the built-in discovery provided by the underlying [DDS] compliant implementation. It does not use any extensions to the DDS discovery built-in topics.

A client comprises one (or more) DataWriter to send requests and one (or more) DataReader to receive replies. Similarly, a service implementation comprises one (or more) DataReader to receive requests and one (or more) DataWriter to send replies. The built-in publication and subscription topics are used to discover the DataWriter and DataReader at both sides.

Non-normative Note: Service discovery for the Basic Service Mapping is not robust because discovery race conditions can cause the service replies to be lost. The request-topic and reply-topic are two different RTPS sessions that are matched independently by the DDS discovery process. For this reason it is possible for the entities on the request topic to discover each other before the entities on the reply topic discover each other. If such a situation, if a client makes a request before the entities over the reply topic are fully discovered, the client may lose the corresponding replies.

7.6.2 Client and Service Discovery for the Enhanced Service Mapping

The Enhanced Service Mapping relies on the built-in discovery service provided by DDS. However, it extends the built-in publication and subscription topic data to avoid the discovery race conditions.

Non-normative Design Rationale: The discovery race condition is avoided by announcing the pair of DataReader and DataWriter endpoints (both client-side and service-side) atomically and ensuring that the discovery of entities over the reply topic is complete before a request is sent or received.

A client must ensure that the client-side DataReader has discovered the service-side DataWriter before writing a request to the service-side DataReader. This ensures that the replies received by the client-side DataReader are not discarded.

A service must ensure that service-side DataWriter has discovered the client-side DataReader before a request is accepted by the service-side DataReader. This ensures that the replies in response to a request are not discarded by the service-side DataWriter.

7.6.2.1 Extensions to the DDS Discovery Builtin Topics

The [DDS] specification includes the ***DCPSPublication*** and ***DCPSSubscription*** builtin topics and corresponding built-in data writers and readers. The data type associated with the ***DCPSPublication*** built-in Topic is `PublicationBuiltinTopicData`. The data type associated with the ***DCPSSubscription*** built-in topic is `SubscriptionBuiltinTopicData`. Both are defined in sub clause 2.1.5 of the [DDS] specification. This specification extends both `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData`.

Non-normative Design Rationale: The extensions specified here are backwards compatible. The [RTPS] specification, which defines the serialization format of `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData`, defines what's called a *ParameterList* where each member in the built-in topic data is serialized using CDR but preceded by a *ParameterID* and *Length* of the serialized member. See sub clause 8.3.5.9 of the [RTPS] specification. This serialization format allows the `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData` to be extended without breaking interoperability.

7.6.2.1.1 Extended PublicationBuiltin TopicData

This specification defines an extension to the `PublicationBuiltinTopicData` structure. The member types and the ParameterID used for the serialization are described below.

Member name	***Member type***	***Parameter ID name***	***Parameter ID value***
service_instance_name	`string<256>`	PID_SERVICE_INSTANCE_NAME	0x0080
related_datareader_key	`GUID_t`	PID_RELATED_ENTITY_GUID	0x0081
topic_aliases	`sequence<string<256>>`	PID_TOPIC_ALIASES	0x0082

```
@extensibility(MUTABLE_EXTENSIBILITY)
struct PublicationBuiltinTopicDataExt : PublicationBuiltinTopicData {
      @ID(0x0080) string<256> service_instance_name;
      @ID(0x0081) GUID_t related_datareader_key;
      @ID(0x0082) sequence<string<256>> topic_aliases;
};
```

The service_instance_name is the instance name specified by the user at the service-side. If none is specified, there is no default value. Likewise, the client-side DataWriter shall not include service_instance_name.

The topic_aliases are as per defined in 7.4.3 The Enhanced Service Mapping. There is no default value.

The related_datareader_key shall be set and interpreted according to the following rules:

- When not present the default value for the related_datareader_key shall be interpreted as GUID_UNKNOWN as defined by sub clause 9.3.1.5 of the [RTPS] specification.

- The PublicationBuiltinTopicDataExt that corresponds to the client-side DataWriter used for sending requests for a service shall have the related_datareader_key set to the BuiltinTopicKey_t of the client-side DataReader used for receiving the replies.

- The PublicationBuiltinTopicDataExt that corresponds to the service-side DataWriter used for sending replies shall have the related_datareader_key set to the BuiltinTopicKey_t of the service-side DataReader used to receive the requests.

7.6.2.1.2 Extended SubscriptionBuiltinTopicData

This specification defines an extension to the SubscriptionBuiltinTopicData structure. The member types and the ParameterID used for the serialization are described below:

Member name	***Member type***	***Parameter ID name***	***Parameter ID value***
service_instance_name	string<256>	PID_SERVICE_INSTANCE_NAME	0x0080
related_datawriter_key	GUID_t	PID_RELATED_ENTITY_GUID	0x0081
topic_aliases	sequence<string<256>>	PID_TOPIC_ALIASES	0x0082

```
@extensibility(MUTABLE_EXTENSIBILITY)
struct SubscriptionBuiltinTopicDataExt : SubscriptionBuiltinTopicData {
     @ID(0x0080) string<256> service_instance_name;
     @ID(0x0081) GUID_t related_datawriter_key;
     @ID(0x0082) sequence<string<256>> topic_aliases;
};
```

The service_instance_name is the instance name specified by the user at the service-side. If none is specified, there is no default value. Likewise, the client-side DataReader shall not include service_instance_name.

The topic_aliases are as per defined in sub clause 7.4.3. There is no default value.

The related_datawriter_key shall be set and interpreted according to the following rules:

- When not present the default value for the `related_datareader_key` shall be interpreted as GUID_UNKNOWN as defined by sub clause 9.3.1.5 of the [RTPS] specification.
- The `SubscriptionBuiltinTopicDataExt` that corresponds to the client-side `DataReader` used to receive replies shall have the `related_datawriter_key` set to the `BuiltinTopicKey_t` of the client-side `DataWriter` used to send the requests to the DDS-RPC Service.

- The `SubscriptionBuiltinTopicDataExt` that corresponds to the service-side `DataReader` used to write replies shall have the `related_datawriter_key` set to the `BuiltinTopicKey_t` of the service-side `DataWriter` used to receive requests.

7.6.2.2 Enhanced algorithm for Service Discovery

The built-in DDS discovery mechanism ensures that DataWriters discover the matching DataReaders and vice versa. However to ensure robustness, service discovery must ensure that no replies are lost due to race conditions in the discovery process of the underlying DDS entities. Additional rules shall be used at the client-side DataWriter and service-side DataReader:

- The client-side DataWriter (request DataWriter) shall not send out any request to a service DataReader unless the client-side DataReader (reply DataReader) has discovered and matched the corresponding service DataWriter (reply DataWriter).
- The service-side DataReader (request DataReader) shall reject requests from a client DataWriter (request DataWriter) as long as the service-side DataWriter (reply DataWriter) has not discovered and matched the corresponding client-side DataReader (reply DataReader).

The above rules may be implemented outside "DDS layer" because no discovery algorithm modifications are required in the underlying DDS implementation to achieve "service discovery" beyond the extensions to the built-in topic data described in 7.6.2.1 Extensions to the DDS Discovery Builtin Topics.

7.6.2.2.1 Client Matching

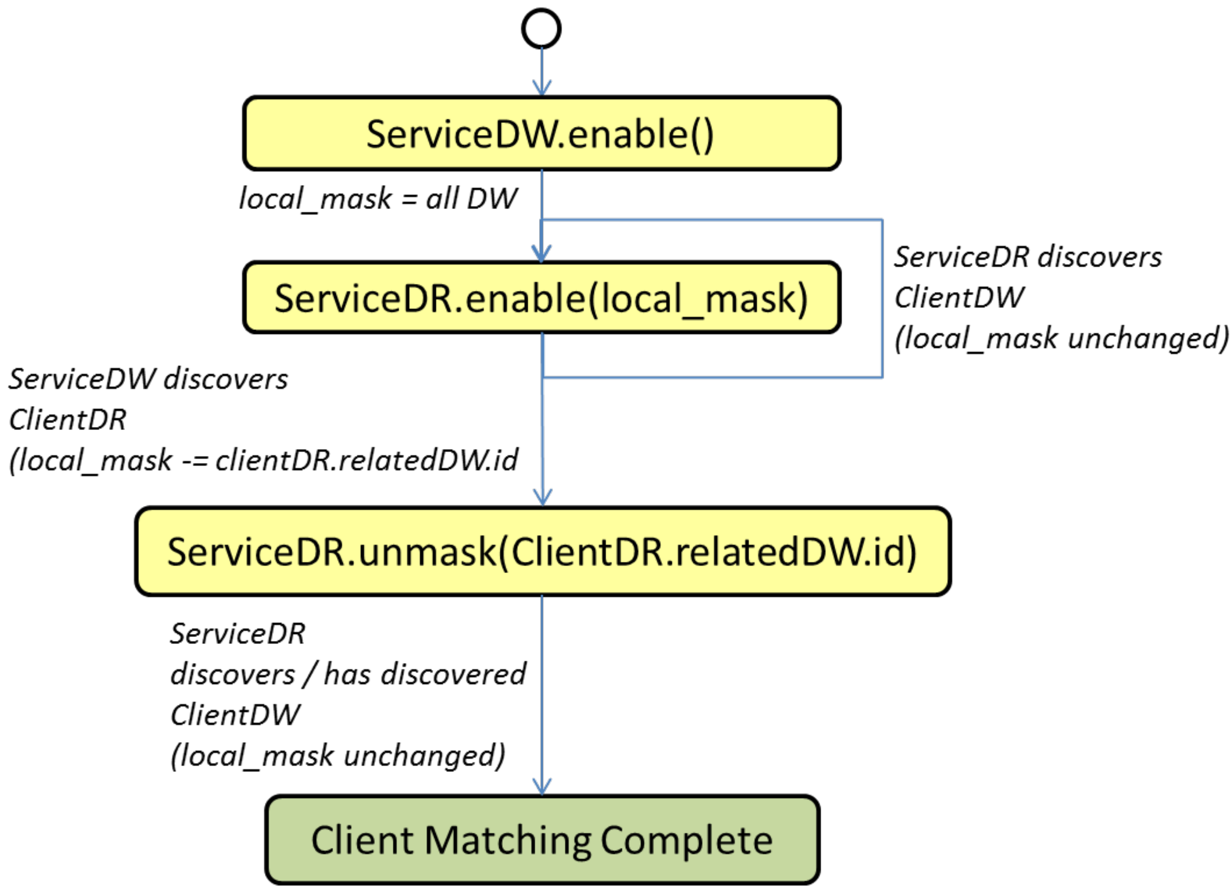

Figure 7.3 - The service-side client matching algorithm

Figure 7.3 shows the client-matching algorithm executed at the service side. Client DataWriter and DataReader are discovered on the service-side using regular DDS discovery but "client matching" does not complete unless the service-side has discovered both client entities (DataReader and DataWriter). While "client matching" is still in progress, service-side does not process requests from the client. This behavior is illustrated in Figure 7.3 using a notional local state called `local_mask`. At the beginning, the service-side DataReader is enabled with `local_mask=All_DW` so that it does not process any incoming requests. When the service-side DataWriter completes the discovery of a client DataReader, the service DataReader is ready to accept requests from that specific client. Therefore, it unmasks the related client DataWriter. The "client matching" algorithm completes when the service-side has discovered both client DDS Entities.

7.6.2.2.2 Service Matching

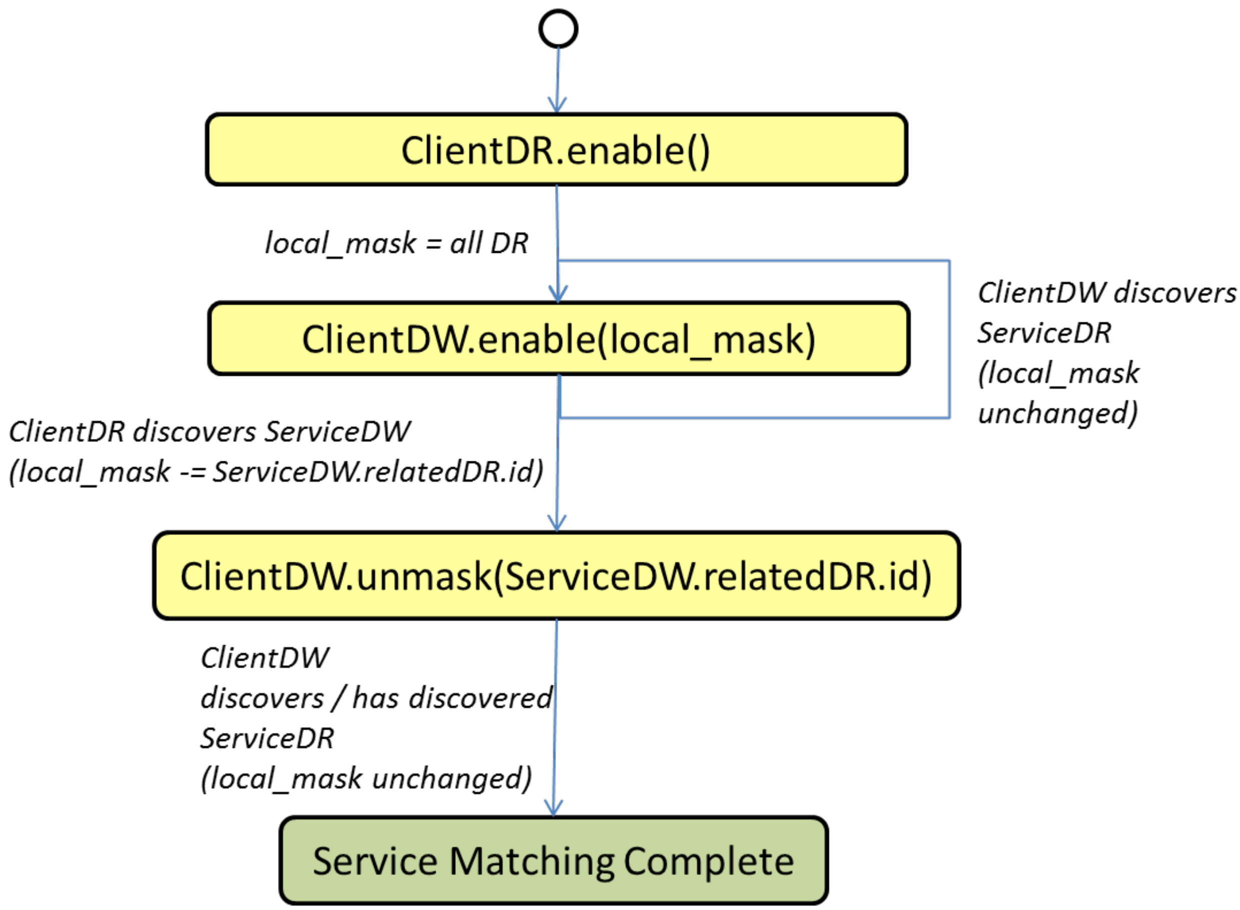

Figure 7.4 - The client-side service matching algorithm

Figure 7.4 shows the service-matching algorithm executed at the client side. Service DataWriter and DataReader are discovered using regular DDS discovery but “service matching” does not complete unless the client-side has discovered both service DDS Entities (DataReader and DataWriter) and it does not send requests to the service until “service matching” completes. This behavior is illustrated in Figure 7.4 using a notional local state called local_mask. At the beginning, the client-side DataWriter is enabled local_mask=All_DR such that it does not send any request at all. Attempts to send requests would be queued. When the client-side DataReader completes the discovery of a service DataWriter, the client DataWriter updates the local mask so that it can send requests to the service DataReader as long as it has been discovered. The “service matching” algorithm completes when the client-side has discovered both the service end-points.

7.7 Interface Evolution

7.7.1 Interface Evolution in the Basic Service Mapping

The Basic Service Profile has limited support for interface evolution (i.e., the interfaces used by the client and the service may not match exactly). Interface evolution is constrained by the rules described herein.

7.7.1.1 Adding/Removing an Operation

A client may use an interface that has fewer or more operations than what the service has implemented. If a client uses an interface that has fewer methods than that of the service, it is simply oblivious of the fact that the service has more capabilities than the client can use. If the client uses an interface with more operations than that of the service, invocation of an unsupported operation shall result in REMOTE_EX_UNSUPPORTED remote exception code.

7.7.1.2 Reordering Operations and Base Interfaces

Reordering operations has no impact on the semantics of the interface because every interface has set semantics. The order in which operations are listed in an interface is irrelevant to the caller. Reordering base interfaces has no impact on the semantics of the interface.

7.7.1.3 Changing Operation Signature

The Basic Service Profile shall not support any modifications to the operation signatures including exception specification.

7.7.2 Interface Evolution in the Enhanced Service Mapping

The Enhanced Service Profile supports flexible interface evolution. It is possible for a client/service pair to use different interfaces where one interface is an evolution of the other. The client may be using an old interface and the service might be using the new interface or vice versa. Interface evolution is constrained by the rules described herein.

In sub clauses that follow, I_{old} represents the old interface whereas I_{new} represents the evolved interface. The interface mapping rules in the previous sub clause are designed such that the assignability rules defined in [DDS-XTypes] will ensure seamless evolution from I_{old} to I_{new}.

Each entity is in one of the four possible roles: C_{old}, C_{new}, S_{old}, S_{new}. C_{old} is a client instance using an older version of the interface. C_{new} is a client instance using the new version of the interface. Likewise, S_{old} is the old version of service implementing the old interface whereas S_{new} is the new instance of the service implementing the new interface.

C_{old} and S_{old} use the topic types synthesized from I_{old} whereas C_{new} and S_{new} use the topic types synthesized from I_{new}. $Request_{old}$ and $Reply_{old}$ are the types synthesized from I_{old} whereas $Request_{new}$ and $Reply_{new}$ are the types synthesized from I_{new}.

7.7.2.1 Adding a new Operation

Adding an operation to an interface will result in additional members in the synthesized topic types. C_{old} can invoke S_{new} because the additional member in $Request_{new}$ is never populated by C_{old}. S_{new} responds with $Reply_{new}$, which remains assignable to $Reply_{old}$ because the ids of members are based on md5 hash and therefore uniquely assignable to the target structure.

When C_{new} invokes S_{old}, S_{old} receives a sample with no active member because the additional member in $Request_{new}$ has no equivalent in $Request_{old}$. In such cases, the service (S_{old}) returns $Reply_{old}$ with `remoteEx =`

REMOTE_EX_UNKNOWN_OPERATION. As the ids of the remoteEx member match in $Reply_{old}$ and $Reply_{new}$, the sample is assignable and the C_{new} receives the correct remote exception code.

7.7.2.2 Removing an Operation

The case of removing an operation from an interface is quite analogous to adding an operation to an interface. When C_{old} calls S_{new}, C_{old} receives REMOTE_EX_UNKNOWN_OPERATION RemoteExceptionCode. Likewise, When C_{new} invokes S_{old}, C_{new} uses only a subset of operations supported by S_{old}.

7.7.2.3 Reordering Operations and Base Interfaces

Reordering operations has no impact on the semantics of the interface because every interface has set semantics. The order in which operations are listed in an interface is irrelevant to the caller. Due to the use of md5 hash algorithm to compute the member ids, this specification considers two interfaces equivalent as long as they contain the same operations. Reordering base interfaces has no impact on the semantics of the interface.

7.7.2.4 Duck Typing Interface Evolution

Enhanced Service Profile supports "duck typing" evolution of an interface.

[**Non-Normative Note:** Duck typing is a style of dynamic typing in which an object's operations and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface. When two interfaces with the same name have the identical set of operations but in one interface all the operations are defined in-place whereas in the other interface one or more operations are inherited, the two (derived-most) interfaces are identical for the purpose of this specification. It is possible to specify the same topic name for two different interfaces. Therefore, as long as the topic names and type names match, the two interfaces are compatible.]

7.7.2.5 Changing Operation Signature

7.7.2.5.1 Adding and Removing Parameters

Adding and removing parameters in an operation shall break compatibility with the older interface. The @final annotation on the *In* and *Out* structures enforce this restriction. It is, however, possible to mark an operation @mutable to allow the *In* and *Out* structures to be mutable and thereby supporting extensions.

7.7.2.5.2 Reordering Parameters

Reordering parameters in an operation shall break compatibility with the older interface. The @final annotation on the *In* and *Out* structures enforce this restriction. It is, however, possible to mark an operation @mutable to allow reordering of parameters. In that case the synthesized structures (*In* and *Out*) shall be annotated @mutable.

7.7.2.5.3 Changing the Type of a Parameter

Changing the type of the parameters is allowed by this specification. The type assignability rules shall be as described in [DDS-XTypes].

7.7.2.5.4 Adding and Removing Return Type

Adding and removing return type shall break compatibility with the older interface. The @final annotation on the *In* and *Out* structures enforce this restriction. It is, however, possible to annotate an operation @Mutable to allow the *In* and *Out* structures to be mutable and consequently support addition or removal of return type and the interface level.

7.7.2.5.5 Changing the Return Type

Changing the type of the return value is allowed by this specification. The type assignability rules shall be as described in [DDS-Xtypes].

7.7.2.5.6 Adding and Removing an Exception

This specification supports adding and removing one or more exceptions on an operation. The `@mutable` annotation on the return structure will support this interface evolution. The language binding that provides function call/return semantics may throw `REMOTE_EX_UNKNOWN_EXCEPTION` if the RPC call results into an exception that client does not understand.

7.8 Request and Reply Correlation

7.8.1 Request and Reply Correlation in the Basic Service Profile

The Basic service profile uses the `RequestHeader` and `ReplyHeader` described in sub clause 7.5.1.1.1.

7.8.2 Request and Reply Correlation in the Enhanced Service Profile

To support propagation of the sample identity of a related request this specification extends the [RTPS] specification adding a new parameter id *PID_RELATED_SAMPLE_IDENTITY* with value 0x0083. This parameter may appear in the inlineQos sub-message element of the DATA sub-message (see sub clause 9.4.5.3 in [RTPS]).

The *PID_RELATED_SAMPLE_IDENTITY* shall be used for data samples sent as reply to a request. When present, it shall contain the CDR serialization of the SampleIdentity structure defined below. The RELATED_SAMPLE_IDENTITY_INVALID constant is used to indicate an invalid/non-existing sample identity.

The value of the *PID_RELATED_SAMPLE_IDENTITY* in a reply message shall be identical to the sample identity (GUID, SN) of the request sample that triggered the reply.

```
struct SampleIdentity {
      GUID_t writer_guid;
      SequenceNumber_t sequence_number;
}
```

RELATED_SAMPLE_IDENTITY_INVALID is defined as { GUID_UNKNOWN , SEQUENCENUMBER_UNKNOWN } both of which are defined in the [RTPS] specification.

7.8.2.1 Retrieve the Request Identity at the Service Side

This specification provides language bindings (7.11 Language Bindings) to retrieve the sample identity of the request in the form of a `SampleIdentity` object.

7.8.2.2 Retrieve the Request Identity at the Client Side

This specification provides language bindings (7.11 Language Bindings) to retrieve the related sample identity of the reply from the reply `SampleInfo` in the form of a `SampleIdentity` object.

7.8.2.3 Propagating Request Sample Identity to the Client

Before sending a reply, the service implementation needs to set the related sample identity of the reply sample. The related sample identity is the same as the identity of the request sample. This specification provides language bindings to set the related sample identity when sending the reply to a request.

7.9 Service Lifecycle

7.9.1 Activating Services

In Basic and Enhanced Service Mappings, when using the function-call style language binding, creating an instance of the `Service` type activates the service. Likewise, when using the request/reply style language binding, creating an instance of a `Replier` activates the service.

Service activation shall result in creation of the underlying DataReader and DataWriter entities and they shall be enabled. An activated service shall be discoverable.

7.9.2 Processing Requests

7.9.2.1 Processing Requests using Function-Call Style Language Binding

The function-call style language binding provides a synchronous way to receive requests and return replies. Concrete implementations of the interface operations shall be invoked as a consequence of receiving requests. The operations must return a reply consistent with the language binding.

7.9.2.2 Processing Requests using Request/Reply Style Language Binding

The request/reply style language binding provides three mutually exclusive ways to process requests.

1. A synchronous service listener callback can be installed during service activation. The synchronous callback must return the reply before the callback returns. It is invoked for each received request.
2. An asynchronous service listener callback can be installed during service activation. The asynchronous callback does not return the reply. The callback receives a handle to the `Replier` instance and the `Replier` API can be used to retrieve the requests and send the replies. Unlike synchronous Replier listener callback, the asynchronous Replier listener allows decoupling of request reception from request processing.
3. The `Replier` API may be used to retrieve the requests and send replies.

7.9.3 Deactivating Services

Deleting (subject to language binding) the service instance deactivates the service. In Java language binding, `Closeable.Close()` deactivates the service. In C++ language binding, the destructor of the service implementation shall deactivate the service. Deactivation shall delete the underlying DDS entities (i.e., DataReader and DataWriter only).

7.10 Service QoS

Both Basic and Enhanced Service Mappings allow QoS annotation with an interface.

7.10.1 Interface QoS Annotation

To set the specific Qos for an interface, the annotation @Qos shall be used. It is defined as follows.

```
@annotation
local interface Qos
{
```

```
  attribute string RequestProfile;
  attribute string ReplyProfile;
};
```

The profile attributes are string URLs of the form: `<protocol>://path/resource` where `protocol`, ":", "//" and `path` are optional. Support for the `file://` protocol is mandatory. If the `protocol://` is omitted, the string attributes are interpreted as if the protocol is `file://`. The URL may end with an optional library name. The profile filename and library_name are separated by a #. For example, `file://path/to/filename#library_name`.

The QoS profiles are XML QoS Profiles as defined in the [DDS4CCM] specification.

7.10.2 Default QoS

When no QoS is specified, default QoS are in effect. The default QoS for the endpoints (DataReaders and DataWriters) are specified as follows. The users may modify the mutable QoS policies at run-time.

QoS Policy	Value
Reliability	DDS_RELIABLE_RELIABILITY_QOS
History	DDS_KEEP_ALL_HISTORY_QOS
Durability	DDS_VOLATILE_DURABILITY_QOS

7.11 Language Bindings

Figure 7.5 shows the relationship of select RPC entities defined by the language bindings.

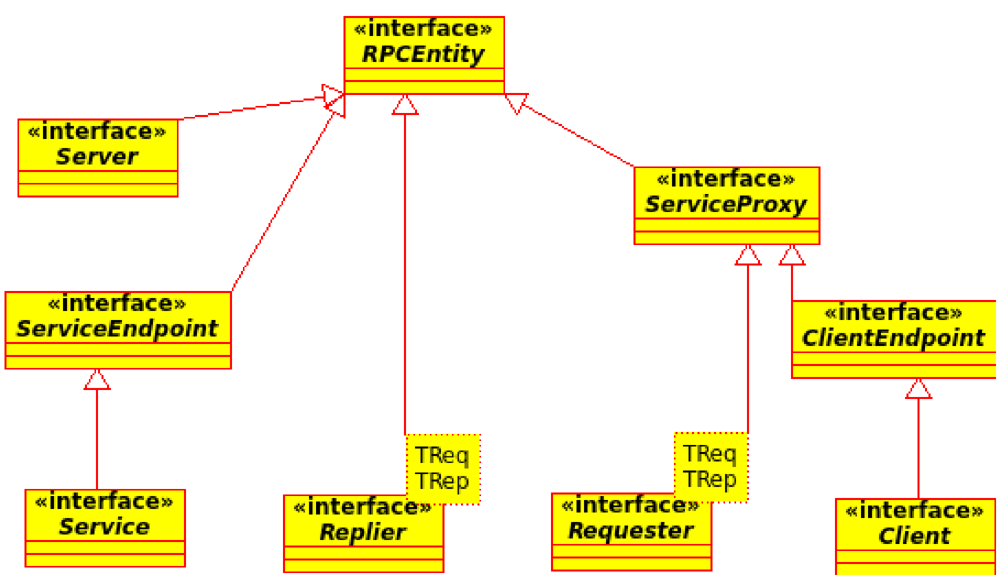

Figure 7.5 - Inheritance Hierarchy of RPC Entities (TReq, TRep are type parameters)

7.11.1 C++ Language Binding

The machine readable files associated with this specification represent the normative language bindings for C++. The following sub clause summarizes the normative reference.

7.11.1.1 General C++ Language Binding Rules for Basic Service Mappings

7.11.1.1.1 Request-Reply Style Language Binding

The request-reply style language binding for the Basic Service Mapping shall use either the user-defined IDL `structs` directly or the `structs` synthesized from an IDL interface. The IDL shall be mapped to C++ types as defined in [IDL35] (including the vendor-specific mappings for DDS v1.3). The `Requester, Replier, Sample`, `SampleRef`, `WriteSample`, and `WriteSampleRef` templates provide a uniform interface to send, receive, allocate, initialize, and destroy data.

7.11.1.1.2 Function-Call Style Language Binding

The function-call style language binding for the Basic Service Mapping shall use the `structs` (and `unions`) synthesized from the user-defined IDL interface as specified in 7.5 Mapping Service Specification to DDS Topics Types. The topic-types shall be mapped to C++ types as per defined in [DDS-Cxx-PSM] with the following additional rules:

1. Each IDL interface shall map to an abstract class named "`${interface}`" and an abstract class named "`${interface}Async`" in the same namespace corresponding to the module of the interface. If there is no module, no C++ namespace is generated.
2. Both "`${interface}`" and "`${interface}Async`" abstract classes shall have a public virtual destructor.
3. Both "`${interface}`" and "`${interface}Async`" abstract classes shall have public `SupportType`, `RequestType`, and `ReplyType` typedefs for "`${interface}Support`," "`${interface}_Request`," and "`${interface}_Reply`" types.
4. The "`${interface}`" abstract class shall contain a public `AsyncInterfaceType` typedef for "`${interface}Async`."
5. The "`${interface}Async`" abstract class shall contain a public `InterfaceType` typedef for "`${interface}`."
6. The "`${interface}`" abstract class shall contain public pure virtual functions for all the operations and attributes defined in the IDL using the following rules:
    a. The name of the function shall be same as the name of the IDL operation.
    b. The number and the order of the arguments shall be as defined in the IDL.
    c. The functions shall not have any exception specification even if the IDL operation has an exception specification. The implementation may still throw the exceptions specified by the IDL operation.
    d. The mapping of IDL primitive and container types to C++ types is provided in sub clause 7.4.2 in [DDS-Cxx-PSM].
    e. The mapping of *In*, *Out*, and *InOut* primitive and constructed types (e.g., `struct`) is provided in sub clause 7.4.5 in [DDS-Cxx-PSM].
    f. Getter/Setter functions for attributes in an interface is specified in sub clause 7.4.6 in [DDS-Cxx-PSM].
    g. IDL operations that return primitive types and enumeration types the corresponding C++ function shall return the C++ type by value as per the mapping specified in sub clause 7.4.2. in [DDS-Cxx-PSM].
    h. IDL operations that return constructed type (e.g., `struct`) shall map to the first parameter with name "`cxx_return`" of type a `T&` to the function and the function shall return `void`.

7. The "`${interface}Async`" abstract class shall contain asynchronous public pure virtual functions for all the operations and attributes defined in the IDL using the following rules:
   a. The name of the function shall be "`${operation}_async`".
   b. The function shall accept only *In* and *InOut* parameters and shall have no *Out* parameters.
   c. The functions shall not have any exception specification even if the IDL operation has an exception specification.
   d. The mapping of IDL primitive and container types to C++ types is provided in sub clause 7.4.2 in [DDS-Cxx-PSM]. All the arguments shall be `const`.
   e. The mapping of *In* and *InOut* primitive and constructed types (e.g.,`struct`) is provided in sub clause 7.4.5 in [DDs-Cxx-PSM]. All the arguments shall be `const`.
   f. Attribute getter functions shall take no parameters and shall return `dds::rpc::future` of the same type as the attribute.
   g. Attribute setter functions shall take a parameter as per specified in sub clause 7.4.6 in [DDS-Cxx-PSM]. It shall return a `dds::rpc::future<void>`.
   h. Operation that produce a return value in the form of a regular return (primitive or constructed type) shall return a value of `dds::rpc::future` of that type.
   i. Operation that produces return values with one or more `Out/InOut` parameters shall return a value of `dds::rpc::future` of "`${interface}_${operation}_Out`" type. Otherwise, it will return `dds::rpc::future<void>`.
8. Any identifier (e.g., parameter, operation name, exception, interface, module) in an IDL file that is a reserved word in C++ shall be prefixed by "`cxx_`".

7.11.1.2 General C++ Language Binding Rules for Enhanced Service Mappings

The request-reply style language binding for the Enhanced Service Mapping shall use either the user-defined IDL `structs` directly or the `structs` synthesized from an IDL interface. The IDL shall be mapped to C++ types as defined in [DDS-Cxx-PSM].

The function-call style language binding for the Enhanced Service Mapping shall use the mapping rules defined in 7.11.1.1.2 Function-Call Style Language Binding.

7.11.1.3 Mapping of Exceptions

The C++ language binding may throw locally generated and remotely generated exceptions. As such, all DDS exceptions defined in sub clause 7.5.5 in [DDS-Cxx-PSM] are valid exceptions. Additionally, remote exception codes defined in 7.5.2 Mapping of Error Codes are mapped to C++ exceptions as follows.

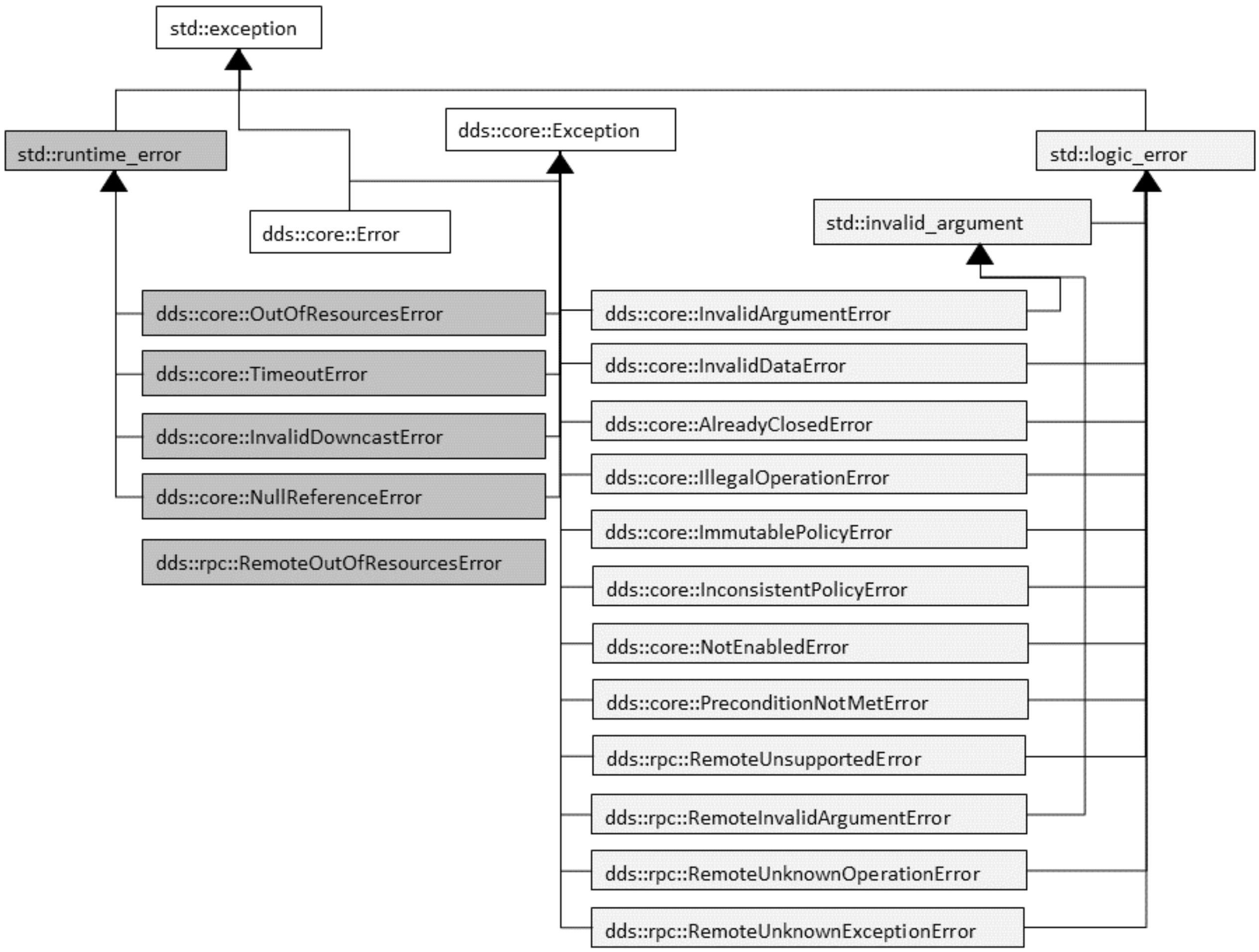

Figure 7.6 - Inheritance Hierarchy of Local and Remote Exceptions

Specifically, the remote exception codes map to C++ exceptions as per the following table.

Remote Exception Code	RPC Exception
REMOTE_EX_OK	No exception
REMOTE_EX_UNSUPPORTED	`dds::rpc::RemoteUnsupportedError`
REMOTE_EX_INVALID_ARGUMENT	`dds::rpc::RemoteInvalidArgumentError`
REMOTE_EX_OUT_OF_RESOURCES	`dds::rpc::RemoteOutOfResourcesError`
REMOTE_EX_UNKNOWN_OPERATION	`dds::rpc::RemoteUnknownOperationError`
REMOTE_EX_UNKNOWN_EXCEPTION	`dds::rpc::RemoteUnknownExceptionError`

7.11.1.4 Summary of C++ Request-Reply Style Language Binding

7.11.1.4.1 Namespaces

The `dds`, and `dds::rpc` namespaces define the classes and functions for the request-reply style language bindings. Specifically, the language binding includes `RPCEntity, Requester, Replier, ServiceProxy, ListenerBase, SimpleReplierListener, ReplierListener, SimpleRequesterListener, RequesterListener, RequesterParams, ReplierParams, future, shared_future, Sample, SampleRef, WriteSample, WriteSampleRef, LoanedSamples, SharedSamples, SampleIterator, dds_type_traits,` and `dds_entity_traits`.

C++ request-reply language binding uses templates. Unless otherwise stated in the following sub clause, `TReq` represents the top-level request type and the `TRep` represents the top-level reply type. Depending upon the profile in use (Basic/Enhanced), the actual structure of the `TReq` and `TRep` types will vary. However, the request-reply language binding is independent of the profile in use.

7.11.1.4.2 RPCEntity

`RPCEntity` is the base abstract class extended by all the active RPC entities. It supports `close()` and `is_null()` operations.

7.11.1.4.3 Requester<TReq, TRep>

A requester sends requests and receives replies. `Requester` is a reference type and when copied it makes a shallow copy. 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.

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.

7.11.1.4.4 ServiceProxy

`ServiceProxy` class defines type-independent operations for `Requester` and the `Client`. For example, binding to a specific instance, waiting for an instance to discover, closing the service, and more. `ServiceProxy` shall not be instantiated directly.

7.11.1.4.5 Replier<TReq, TRep>

A replier receives requests and send replies. Replier is a reference type and it is inexpensive to copy (comparable to a pointer assignment). 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.

Replier allows listener-based and polling-based reception of requests. `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.

7.11.1.4.6 ListenerBase

ListenerBase is a “marker” abstract base class, which all generic listener abstract classes inherit from. It defines no member functions.

7.11.1.4.7 SimpleReplierListener<TReq, TRep>

SimpleReplierListener<TReq, TRep> is used to provide a synchronous request listener for a Replier. ReplierParams is used to pass an instance of SimpleReplierListener<TReq, TRep>. It extends the ListenerBase abstract class and enables synchronous processing of the requests. I.e., the callback provides the request object and expects the reply as the return value.

7.11.1.4.8 ReplierListener<TReq, TRep>

ReplierListener<TReq, TRep> interface is used to provide an asynchronous request listener for a Replier. It is passed to the Replier constructor through ReplierParams. It extends ListenerBase interface and enables asynchronous processing of the requests (i.e., the callback provides the Replier object and returns void).

7.11.1.4.9 SimpleRequesterListener<TRep>

SimpleRequesterListener<TRep> abstract class is used to receive notifications of arrival of replies. It is used to configure a Requester through RequesterParams. It extends ListenerBase and enables processing of replies. I.e., the callback provides access to the reply sample and returns void.

7.11.1.4.10 RequesterListener<TReq, TRep>

RequesterListener<TReq,TRep> interface is used to receive notification of reply arrival. It is used to configure a Requester through RequesterParams. It extends ListenerBase and enables processing of replies. I.e., the callback provides the replier object and returns void.

7.11.1.4.11 RequesterParams

RequesterParams is a valuetype that serves as a container of configuration parameters of a Requester. It is designed to mimic the named-parameters feature available in some programming languages, which improves readability.

7.11.1.4.12 ReplierParams

ReplierParams is a valuetype that serves as a container of configuration parameters of a Replier. It is designed to mimic the named-parameters feature available in some programming languages, which improves readability.

7.11.1.4.13 future<T>

future<T> provides a mechanism to access the result of an asynchronous operation. It transports results (including exceptions) across an asynchronous boundary. In C++11 and C++14 environments, dds::rpc::future<T> is a typedef for std::future<T>. Compiler prior to C++11, dds::rpc::future<T> shall provide the same API as std::future<T> in C++11. Future is not copyable but it is movable.

7.11.1.4.14 shared_future<T>

shared_future<T> is closely related to future<T> and the only difference is that shared_future<T> is copyable and movable.

7.11.1.4.15 Sample<T>

Sample<T> is a valetype that combines a value of type T and a value of type SampleInfo. Sample is conceptually immutable.

7.11.1.4.16 SampleRef<T>

SampleRef<T> is a reference type that combines a value of type T and a value of type SampleInfo. Copying SampleRef<T> makes a shall copy.

7.11.1.4.17 WriteSample<T>

`WriteSample<T>` is a valuetype that combines a value of type `T` and a value of type `dds::SampleIdentity`. The user populates the value of `T` and the middleware populates the value of `dds::SampleIdentity`. When a request is sent as a `WriteSample`, upon function return, the `WriteSample.identity()` uniquely identifies the request sent.

7.11.1.4.18 WriteSampleRef<T>

`WriteSampleRef<T>` is a reference type, which groups a reference to `T` and a value of type `dds::SampleIdentity`. The user populates the value of `T` and the middleware populates the value of `dds::SampleIdentity`. When a request is sent as a `WriteSampleRef<T>`, upon function return, the `WriteSample.identity()` uniquely identifies the request sent.

7.11.1.4.19 LoanedSamples<T>

`LoanedSamples<T>` is conceptually a container of loaned `Sample<T>` from the middleware. `LoanedSamples<T>` is not copyable but it is movable. `LoanedSamples<T>::value_type` is `SampleRef<T>`. Upon destruction, `LoanedSamples<T>` returns the loaned samples to the middleware.

7.11.1.4.20 SharedSamples<T>

`SharedSamples<T>` is container that contains loaned samples from the middleware but the `SharedSamples<T>` object may be copied in the application space. All copies of a `SharedSamples<T>` refer to the same set of loaned samples. `SharedSamples<T>::value_type` is `SampleRef<T>`. Upon destruction of the last `SharedSamples<T>`, the underlying loaned samples are returned to the middleware. Note that `SharedSamples<T>` can be obtained from `LoanedSample<T>` but not vice versa.

7.11.1.4.21 SampleIterator<T>

SampleIterator<T> is a random-access iterator over LoanedSamples<T> and SharedSamples<T>. SampleIterator<T>::value_type is a SampleRef<T>.

7.11.1.4.22 dds_type_traits<T>

`dds_type_traits<T>` is a collection of meta-functions that given a type `T`, provides commonly needed dependent types, such as a DataReader for `T`, DataWriter for `T`, `Sample<T>`, `LoanedSamples<T>`, etc. The primary use of `dds_type_traits` is to provide a consistent syntax to refer to the dependent types irrespective of the DDS C++ language binding.

7.11.1.4.23 dds_entity_traits

`dds_entity_traits` abstracts over the DDS entity types and provides consistent syntax to get DDS entity types irrespective of the DDS C++ language bindings. (e.g., `DomainParticipant`, `Publisher`, `Subscriber`, `DataReaderQos`, and `DataWriterQos`).

7.11.1.5 Summary of C++ Function-Call Style Language Binding

`Service`, `ServiceEndpoint`, `Server`, `Client`, and `ClientEndpoint` classes are specific to the function-call style language binding and are described below.

7.11.1.5.1 Service

`Service` is a reference type and accepts a reference to the service implementation. A `Service` instantiates the underlying DDS entities and makes the service discoverable. Every `Service` inherits from `ServiceEndpoint`. A `Service` may belong to only one `Server`.

7.11.1.5.2 ServiceEndpoint

A `ServiceEndpoint` provides type-independent functions to manage a service (e.g., `pause`, `resume`, `close`, etc.) A `ServiceEndpoint` shall not be instantiated directly; it can be obtained from a `Service` object. `ServiceEndpoint` is a reference type.

7.11.1.5.3 Server

A `Server` is a container of one or more Services. `Server.run()` function begins dispatching the requests to the `Service` implementation. `Server` is a reference type.

7.11.1.5.4 Client

`Client` is a reference type and provides functions to invoke operations on a remote service synchronously and asynchronously. A `Client` instantiates the underlying DDS entities and makes them discoverable. Every `Client` inherits publicly from "`${interface}`", "`${interface}Async`", and `ClientEndpoint` (public virtual).

7.11.1.5.5 ClientEndpoint

A `ClientEndpoint` provides functions to obtain the underlying DDS entities at the client side. `ClientEndpoint` inherits from `ServiceProxy`. A `ClientEndpoint` shall not be instantiated directly; it can be obtained from a `Client` object.

7.11.2 Java Language Binding

The machine readable files associated with this specification represent the normative language bindings for Java. The Java language binding shall use IDL type mapping as per [DDS-Java-PSM].

The following sub clauses describe the entities that are either different or not described in the C++ language bindings.

7.11.2.1 Mapping of Exceptions

7.11.2.2 Summary of Java Request-Reply Style Language Binding

7.11.2.2.1 Packages

The `org.omg.dds,` and `org.omg.dds.rpc` packages define the interfaces for the request-reply style language binding. Specifically, the language binding includes `Requester, Replier, ServiceProxy, RPCEntity, RPCRuntime, SimpleReplierListener, ReplierListener, SimpleRequesterListener, RequesterListener, RequesterParams, ReplierParams, Future, FutureCompletionListener, Sample,` and `Sample.Iterator`.

7.11.2.2.2 RPCEntity

`RPCEntity` is the base interface extended by all the active entities. It inherits from the `DDSObject` interface defined in [DDS-Java-PSM] and `java.io.Closeable`. Its purpose is to provide quick access to the `RPCRuntime` object.

7.11.2.2.3 RPCRuntime

`RPCRuntime` is the only abstract class with placeholder implementation in the Java language binding. It extends `ServiceEnvironment` abstract class defined in [DDS-Java-PSM]. Its purpose is to provide access to the `RPCRuntime` singleton, which serves as a factory for all other entities.

7.11.2.2.4 Future<T>

The `rpc.Future<T>` interface extends the `java.util.concurrent.Future<T>` interface and adds a single operation to specify a `FutureCompletionListener`.

7.11.2.2.5 FutureCompletionListener<T>

The `FutureCompletionListener` allows callback notification when the corresponding future becomes ready.

7.11.2.2.6 Sample<T>

The `rpc.Sample<T>` interface extends `dds.sub.Sample<T>` interface and defines two additional operations to retrieve the `SampleIdentity` and the related sample identity, if any.

7.11.2.2.7 Sample.Iterator

The `Sample.Iterator` interface is the same as in [DDS-Java-PSM].

7.11.2.3 Summary of Java Function-Call Style Language Binding

Function-Call style language binding in Java shall define the same entities as that of C++. See 7.11.1.5 Summary of C++ Function-Call Style Language Binding.