Date: May 2014

Remote Procedure Call over DDS

Submitters:

Real-Time Innovations, Inc. (RTI)

eProsima

Supporters:

Twin Oaks Computing, Inc.

OMG Document Number: **mars/2014-05-05**
Normative reference: **http://www.omg.org/cgi-bin/doc?mars/2014-05-05**
Machine readable file (C++): **http://www.omg.org/cgi-bin/doc?mars/2014-05-06**
Machine readable file (Java): **http://www.omg.org/cgi-bin/doc?mars/2014-05-07**

In response to OMG Request for Proposal (RFP):
Remote Procedure Calls (RPC) over Data Distribution Service (DDS) mars/2012-06-29

Submission contacts:

Sumant Tambe, Ph.D. (lead)
Real-Time Innovations, Inc.
sumant@rti.com

Gearrdo Pardo-Castellote, Ph.D.
Real-Time Innovations, Inc.
sumant@rti.com

Jaime Martin-Losa
eProsima
JaimeMartin@eProsima.com

Clark Tucker
Twin Oaks Computing, Inc
ctucker@twinoakscomputing.com

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

DISCLAIMER OF WARRANTY

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

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 (http://www.omg.org/technology/agreement.)

Table of Contents

Contents

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

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:

Object Management Group
109 Highland Ave
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 section headings where no distinction is necessary.
Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.
`Courier - 10 pt. Bold:` Programming language elements.
Helvetica/Arial - 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to http://www.omg.org/report_issue.htm.

PART I - Introduction

1 Introduction

1.1 Name of the submission

Remote Procedure Calls (RPC) over Data Distribution Service (DDS)

1.2 Contact

Real-Time Innovations, Inc. – Sumant Tambe. sumant@rti.com
eProsima .- Jaime Martin-Losa JaimeMartin@eProsima.com

1.3 OMG RFP Response

This specification is submitted in response to the Remote Procedure Calls (RPC) over Data Distribution Service (DDS) MARS/2012-06-29.

1.4 Copyright Waiver

1.5 Overview and Design Rationale

The Data Distribution Service is widely used for data-centric publish/subscribe communication in real-time distributed systems. The DDS is used across a variety of application domains including Aerospace and Defense, Transportation, SCADA, and Financial. 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 submission presents a solution to this problem. Specifically, it is a response to the Remote Procedure Calls (RPC) over DDS RFP issued by OMG.

1.6 Statement of Proof of Concept

The design of the technical submission is based on RTI and eProsima current products called “RTI Connext Messaging” and “eProsima RPC over DDS”

RTI Connect Messaging provides basic facilities to create scalable request/reply communication pattern. For more information please see http://www.rti.com/products/messaging/index.html
eProsima RPC over DDS enables RPC over any DDS compliant implementation. For more information please visit www.eprosima.com

PART II – RPC over DDS

2 Scope

This submission defines a Remote Procedure Calls (RPC) framework using the basic building blocks of DDS, such as topics, types, and entities (e.g., DataReader, DataWriter) to provide request/reply semantics. It defines distributed services, characterized by 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.

3 Conformance

This specification defines two conformance points: Basic and Enhanced.

[1] Basic conformance (mandatory).

[2] Enhanced conformance (optional).

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

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

4 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/IIOP). formal/2011-11-01, formal/2011-11-02, and formal/2011-11-03
- **[IDL]** Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 1 (OMG document formal/2008-01-04), section 7: "OMG IDL Syntax and Semantics"
- **[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-CPP-PSM**] C++ Language PSM for DDS (OMG document ptc/2012-10-01)
- [**DDS-Java-PSM**] Java 5 Language PSM for DDS (OMG document ptc/2012-12-02)
- **[DDS-LWCCM]** DDS for lightweight CCM
- **[IDL2C++11]** IDL To C++11 Language Mapping, Version 1.0 http://www.omg.org/spec/CPP11/1.0/
- **[IDL2Java]** IDL to Java Language Mapping, version 1.3 http://www.omg.org/spec/I2JAV/1.3/
- **[DDS-RPC-CXX]** C++ API for RPC over DDS, https://code.google.com/p/dds-rpc-cxx/
- **[ddsrpc4j]** Java API for RPC over DDS, https://code.google.com/p/ddsrpc4j/

5 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 consistent with constraints and policies as specified by the service description. (Source: OASIS SOA Reference Model)

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

6 Symbols

DDS	Data-Distribution Service
GUID	Global Unique Identifier
RPC	Remote Procedure Call

RTPS Real-Time Publish-Subscribe Protocol

SN Sequence Number

7 Additional Information

7.1 Changes to Adopted OMG Specifications

None

7.2 Acknowledgements

The following companies submitted this specification:

- Real-Time Innovations, Inc.
- eProsima

The following companies supported this specification:

- Real-Time Innovations, Inc.
- eProsima

8 Remote Procedure Call over Data Distribution Service

8.1 Overview

Large distributed systems often require different communication styles depending upon the problem at hand. For instance, distribution of sensor data is best achieved using unidirectional *one-to-many* style whereas sending commands to a specific device or retrieving configuration of a remote service is best done using bidirectional request/reply style 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. 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 submission 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 submission 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.

8.2 General Concepts

8.2.1 Architecture

Remote Procedure Call necessarily has two key 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 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 “message-id” (a.k.a sample identity) defined as the tuple of DataWriter GUID and Sequence Number (SN). 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.

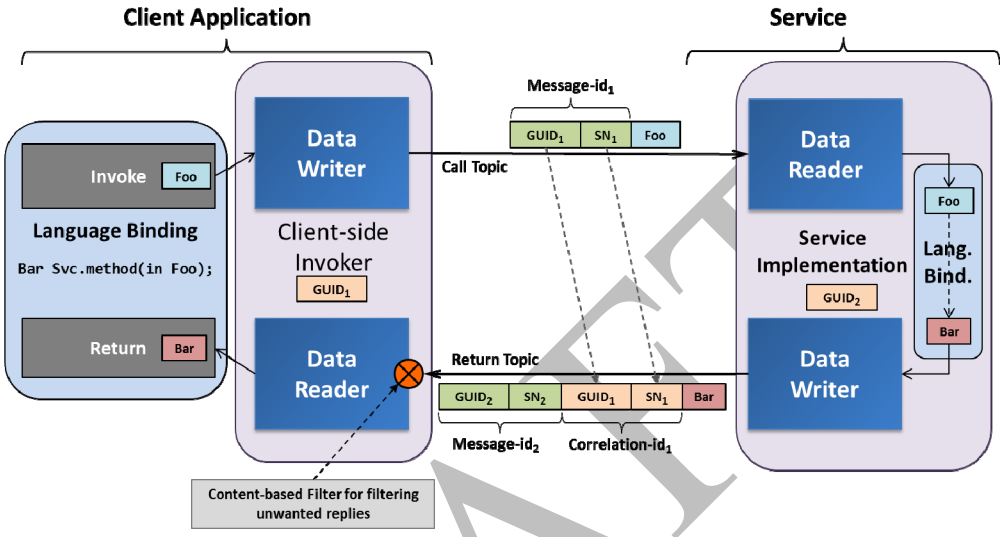

Figure 1: Conceptual View of Remote Procedure Call over DDS

8.2.2 Language Binding Style 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 classifies language binding style into two broad categories: (1) higher-level language binding with *function-call* style and (2) lower-level language binding with *request/reply* style.

8.2.2.1 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 proxies for remote operations and *skeletons* to support service-side implementations. The look-and-feel is like a local function invocation. The 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. However, complex parameter passing issues and special handling of asynchronous calls must be addressed if this approach is chosen.

8.2.2.2 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 requests and replies. The programmer

is responsible to populate 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 *call*, *receive*, and r*eply,* which substantially simplifies language binding and asynchronous service invocation. However remote procedure call does not appear *first-class* to the programmer.

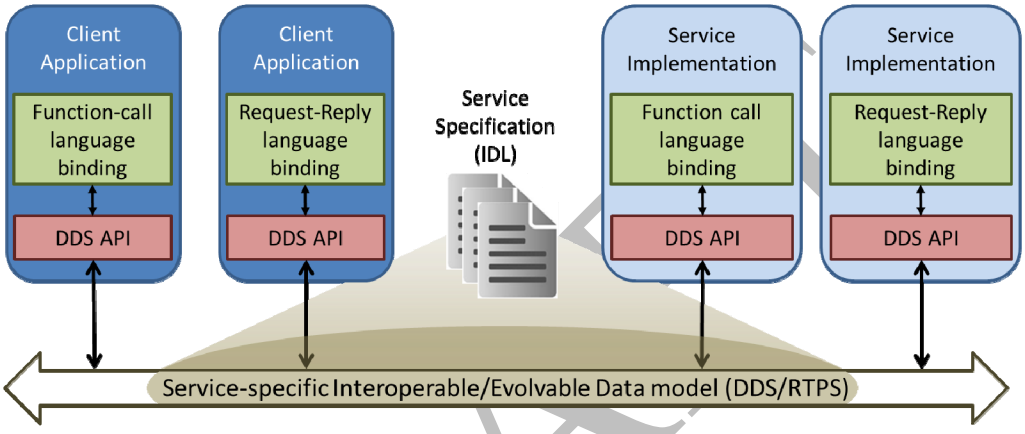

Figure 2: Strong decoupling of language binding (function-call and request/reply) from the service-specific data model

8.2.2.3 Pros and Cons of each Language Binding Styles

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 service *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 inverse scenario is also allowed. Furthermore, it is also possible and the intent of this specification to layer the stubs and skeletons of the functional style to use the request/reply language binding style under the hood. See Figure 2.

In light of the above observation, this specification requires that every conforming implementation must support the both styles of language binding.

8.2.3 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 request-id (and meta-information) can be propagated either *implicitly* or *explicitly*. Explicit meta-information implies that the request-id is visible in the top-level data type for the DDS topic. Implicit meta-information, on the other hand, implies that the request sample idenity is communicated via extensibility mechanisms supported by the DDS-RTPS protocol. Specifically the inlineQoS sub-message element . See Section 8.3.7.2 of the DDS-RTPS specification [2].

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.

8.2.4 Basic and Enhanced Service Mapping for RPC over DDS

This specification describes two service mappings for interfaces to DDS topics and types. These mappings address different system needs. The two service mappings are 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. They appear 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 independently of interface hierarchies.
Type Mapping	Synthesized types compatible with legacy (pre DDS-XTypes) implementations.	Use facilities of DDS-XTypes for type descriptions, annotations, and type-compatibility checks. See section XXX
Discovery	No special extensions.	Robust service discovery as described in Section 8.6

Client and service interoperability requires both sides to us 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. I.e., It is possible for clients and service implementations to use different language binding styles as long as the choice of their Service Mappings match.

8.3 Service Definition

Non-normative: OASIS Service-Oriented Architecture reference model defines service as a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.

According to the definition and for the purpose of this specification, a service specification consists of two parts: (1) a specification of the interface, and (2) a specification of the constraints and policies. The interface is defined either using OMG Interface Definition Language (IDL) or Java and the constraints and policies, if any, are specified using annotations.

8.3.1 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. This specification considers that asynchronous invocation and execution are artifacts of the language binding and do not belong in the service specification.

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.

8.3.1.1 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 section 5.4 (IDL Grammar) of the IDL 3.5 [4] and section 7.3.1.12.1 (New Productions) of the DDS-XTypes specification [3].

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.

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

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

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

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

(87) <op_dcl>          ::= [ <annotation> ] <op_type_spec> <identifier> <parameter_dcls>
                           [ <raises_expr> ] [ <context_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 section 7.3.1.12.1 of the DDS-XTypes specification.

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 <annotations> production is defined in section 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 section 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>
(87)	IDL 3.5	Modified to add support for annotations on operations and also remove the CORBA-specific "oneway" modifier
(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
XTypes	DDS-XTypes section 7.3.1.12.2	These productions add support for annotating types and the new IDL types defined in the 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.

8.3.1.2 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 remaining productions in Section 7.3.1.12.1 (New Productions) of the DDS-XTypes specification.

In addition the Enhanced Service Mapping also uses all the modified defined in Section 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.

8.3.1.3 Service Definition in IDL Examples

8.3.1.3.1 Example: Simple RobotControl Interface in IDL

The following example defines a *DDSService* named `RobotControl`.

```
exception TooFast { float maxSpeed; }
```

```
@DDSService
interface RobotControl {
    void start();
    void stop();
    boolean setSpeed(inout float s) raises (TooFast);
    long getSpeed();
};
```

8.3.1.3.2 Example: Calculator Interface in IDL inheriting from other Interfaces

The following example defines a *DDSService* named `Calculator`. That inherits from `Adder` and `Subtractor` interfaces.

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

8.3.2 Service Definition in Java

The BNF grammar used to define the Java syntax uses the in [Java-Grammar]. The Java grammar shall be modified with the productions shown below. In, out, and InOut parameters are supported 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.

```
InterfaceDeclaration:
      NormalInterfaceDeclaration
NormalInterfaceDeclaration:
     interface Identifier [extends TypeList] InterfaceBody
InterfaceBody:
        { {InterfaceBodyDeclaration} }
InterfaceGenericMethodDecl:
        (Type | void) Identifier InterfaceMethodDeclaratorRest
```

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

Production Name	*Explanation*
`InterfaceDeclaration`	Removed `AnnotationTypeDeclaration` because Java annotation type declarations are out of scope for this specification.
`NormalInterfaceDeclaration`	Removed `[ TypeParameters]` because generic interfaces are not within the scope of this specification.
`InterfaceGenericMethodDecl`	Removed `TypeParameters` because generic methods are not within the scope of this specification.

Field (static or otherwise) declarations and empty Java interfaces shall be ignored.

8.3.2.1 Service Definition in Java Example

The following example defines a *DDSService* named `RobotControl` using a Java interface.

```
public class TooFast extends Exception { public float maxSpeed; }
@DDSService
interface RobotControl {
    public void start();
    public void stop();
    public boolean setSpeed(@inout float s) throws TooFast;
    public long getSpeed();
};
```

Similar to the use of IDL, service specification in Java does not specify whether an operation is synchronous or asynchronous. This is determined as part of the language binding.

8.4 Mapping Service Specification to DDS Topics

Each Service Mapping provides three alternative mechanisms to specify the names of the DDS 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.

8.4.1 Basic Service Mapping

The Basic Service Mapping shall map every interface to two DDS Topics. One for the request and one for the reply. 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.

For example the interface `Calculator` defined below would result in six DDS Topics: Two Adder, two for Subtractor, and two for Calculator:

```
interface Adder { ... };

interface Subtractor { ... };

interface Calculator : Adder, Subtractor { ... }
```

8.4.1.1 Default Topic Names

By default, the DDS Topic name shall be synthesized by appending the interface name, an underscore and a "_Request" or "_Reply" suffix. For example, if the interface name is `RobotControl`, the default request topic name shall be "RobotControl_Request" and the default reply topic name shall be "RobotControl_Reply".

8.4.1.2 Specifying Topic Names using Annotations

Annotations may be used at the interface level 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:

```
@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.

For example, the RobotControl interface may use one or both of the annotations shown below.

```
@DDSService
@DDSRequestTopic(name="RobotRequestTopic")
@DDSReplyTopic(name="RobotReplyTopic")
interface RobotControl
{
    void start();
   ...
}
```

8.4.1.3 Specifying Topic Names at Run-time

DDS Topic names may also be specified at run-time. 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.

8.4.2 The Enhanced Service Mapping

The Enhanced Service Mapping shall map an interface to two DDS Topics. One for request and reply topics. In case of an interface inheritance hierarchy, including multiple inheritance, the entire hierarchy shall map to a single request Topic and a single reply Topics. This is different from the Basic Service Mapping.

For example the interface `Calculator` defined below would result in two DDS Topics despite the fact that `Calculator` inherits from other interfaces.

```
interface Adder { ... };

interface Subtractor { ... };

interface Calculator : Adder, Subtractor { ... }
```

8.4.2.1 Default Topic Names

By default, the DDS Topic name shall be synthesized by appending the interface name, an underscore and a "_Request" or "_Reply" suffix. For example, for the RobotControl service above the default request Topic name shall be "RobotControl_Request" and the default reply Topic name shall be "RobotControl_Reply".

To support interface inheritance and polymorphism, all the interfaces in a hierarchy must map to a common topic defined by the user. By default, in the case of single inheritance hierarchy, all the interfaces shall use the topic name obtained from the base interface. No default is specified for interfaces that derive from multiple interfaces.

8.4.2.2 Specifying Topic Names Using Annotations

Annotations may be used at the interface level to specify the names of the *request* and the *reply* topics. The same built-in @DDSRequestTopic and @DDSReplyTopic annotations defined in the Basic Service Mapping may be used for this purpose.

For example, the `RobotControl` service above may specify one or both the topics as follows.

```
@DDSService
@DDSRequestTopic(name="RobotControlIn")
@DDSReplyTopic(name="RobotControlOut")
interface RobotControl { ... }
```

8.4.2.3 Specifying Topic Names at Run-time

Topic names may also be specified at run-time. When more than one method of specifying topic names is used, the run-time specification will take precedence over IDL annotations and the default mechanism.

8.5 Mapping Service Specification to DDS Topics Types

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

8.5.1 Annotations for the Enhanced Service Mapping

Comment [GP1]: I would move this as a subsection of the Enhanced Service Mapping (under 8.6.4.2). It does not fit well at the top level.

8.5.1.1 `@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.

8.5.1.2 `@Autoid` Annotation

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

The `@Autoid` annotation indicates that the memberid of each member shall be computed as the lower 32 bits of the MD5 hash applied to the name of the member. As IDL does no support overloading, no two members will have the same name. Consequently, the memberids 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 memberid 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 for the semantics of an interface. Further, it supports interface evolution (including new operations, operation reorder and new base interfaces) without changing the member ids.

Comment [GP2]: Do we need all this flexibility on the declararation of operations…

8.5.1.3 `@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.

8.5.1.4 `@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.

8.5.2 Common Types for the Basic Service Profile

Comment [GP3]: I would move this as a subsection of the Basic Service Mapping (under 8.6.4.1)

All the generated types as per the Basic Service Profile use a set of common types.

```
typedef octet UnknownOperation;
typedef octet UnknownException;

struct GUID_t {
  octet value[16];
};
struct SampleIndentity_t {
   GUID_t guid;
   long long sequence_number;
};
struct RequestHeader {
  SampleIndentity_t request_id;
  string<255> remote_service_name;
  string<255> instance_name;
};
struct ReplyHeader {
  SampleIndentity_t request_id;
};
```

8.5.3 Mapping of System Exceptions for Basic and Enhanced Profiles

System exception codes is an enumeration as defined below.

```
enum SystemExceptionCode
{
  OK,
  UNIMPLEMENTED,
  BAD_PARAMETER,
  PRECONDITION_NOT_MET,
  OUT_OF_RESOURCES,
  NOT_ENABLED,
```

```
  ALREADY_DELETED,
  ILLEGAL_OPERATION,
  TIMEOUT,
  INITIALIZE_ERROR,
  CLIENT_INTERNAL_ERROR,
  SERVER_INTERNAL_ERROR,
  SERVER_NOT_FOUND,
  UNKNOWN
};
```

`BAD_PARAMETER, PRECONDITION_NOT_MET, OUT_OF_RESOURCES, NOT_ENABLED, ALREADY_DELETED, TIMEOUT, INITIALIZE_ERROR , CLIENT_INTERNAL_ERROR, SERVER_NOT_FOUND, and UNKNOWN` exceptions may be raised locally.

`INITIALIZE_ERROR , SERVER_NOT_FOUND , UNIMPLEMENTED, BAD_PARAMETER, OUT_OF_RESOURCES, UNKNOWN` exceptions may be raised remotely.

8.5.4 Interface Mapping

8.5.4.1 Basic Service Mapping of Interfaces

The Basic Service Mapping maps interfaces to data-types that can be used by legacy DDS implementations that lack support of the DDS-XTypes specification.

8.5.4.1.1 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.

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 name 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 long and name "dummy".

For example, the operations in the `RobotControl` interface defined in section 8.3.1.3.1 shall map to the following *In* structures.

```
struct RobotControl_start_In { long dummy; };
```

```
struct RobotControl_stop_In { long dummy; };

struct RobotControl_setSpeed_In {
  long s;
};

struct RobotControl_getSpeed_In { long dummy; };
```

8.5.4.1.2 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. 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:

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. If the operation defines a return value the *Out* structure shall have a first member named "return_". The type of the "return_" member shall be the return type of the function. In the case where the function does not define a return value this member shall not be present.
    b. The *Out* structure shall one additional 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.
    c. If the operation has no return value, no *inout* parameters and no *out* parameters, the *Out* structure shall contain a single member with name *dummy* and type long.

For example, the operations defined in the `RobotControl` interface defined in section 8.3.1.3.1 map to the following *Out* structures:

```
struct RobotControl_start_Out { long dummy; };
struct RobotControl_stop_Out { long dummy; };

struct RobotControl_setSpeed_Out {
  boolean return_;
  long s;
};

struct RobotControl_getSpeed_Out {
  long return_;
};
```

The following rules define the *Result* union. 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:

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 default case label containing a member named "unknown_exception" of type `UnknownException`.
4. The *Result* union shall have a case with label 0 which is used to represent a successful return.
    a. This case label shall contain a single member with name "out" and type ${interfaceName}_${operationName}_Out. This type is defined in section 8.5.4.1.2.
5. The union shall have a case with label 1 which is used to represent a system exception.
    a. This case shall contain a single member with name "sysx_" and type SystemExceptionCode.
6. 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 the least significant 4 bytes of the MD5 hash of the name of the exception type.
    b. The case label shall contain a single member with name synthetized as the lower-case version of the exception name with the suffix "_ex" added.
    c. The type associated with the case member shall be the exception type..

For example, the operations defined in the `RobotControl` interface defined in section 8.3.1.3.1 map to the following *Result* unions:

```
union RobotControl_start_Result switch(long) {
  default:  UnknownException unknown_exception;
  case 0: RobotControl_start_Out out;
  case 1: SystemExceptionCode_sysx_;
};

union RobotControl_stop_Result switch(long) {
  default:  UnknownException unknown_exception;
  case 0: RobotControl_stop_Out out;
  case 1: SystemExceptionCode_sysx_;
};

union RobotControl_setSpeed_Result switch(long) {
  default:  UnknownException unknown_exception;
  case 0: RobotControl_setSpeed_Out out;
  case 1: SystemExceptionCode_sysx_;
  case HASH("TooFast"):  TooFast toofast_ex;
```

```
};

union RobotControl_getSpeed_Result switch(long) {
  default:  UnknownException unknown_exception;
  case 0: RobotControl_getSpeed_Out out;
  case 1: SystemExceptionCode_sysx_;
};
```

In the above IDL the short-hand HASH(a_string) is used for clarity. In the actual IDL it shall be substituted with the actual value computed according to the formula:

```
octet md5_hash[16];
md5_hash := compute_md5( a_string );

HASH :=                md5_hash[0] +
                   256* md5_hash[1] +
               256*256* md5_hash[2] +
          256*256*256* md5_hash[3];
```

8.5.4.1.3 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. In these rules the token ${interfaceName} shall be replaced with the name of the interface:

1. The *Call* union name shall be "${interfaceName}_Call".
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 "unknown_operation" of type `UnknownOperation`.
4. 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 least-significant 4 bytes of the MD5 hash applied to the operation name.
    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 section 8.5.4.1.1

For example, the `RobotControl` interface defined in section 8.3.1.3.1 shall result in the union `RobotControl_Call` defined below:

```
union RobotControl_Call switch(long) {
  default:
    UnknownOperation unknown_operation;
```

```
  case HASH("start"):
    RoboControl_start_In start;
  case HASH("stop"):
    RoboControl_stop_In stop;
  case HASH("setSpeed"):
    RoboControl_setSpeed_In setSpeed;
  case HASH("getSpeed"):
    RoboControl_getSpeed_In getSpeed;
};
```

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

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

For example, the `RobotControl` interface defined in section 8.3.1.3.1 shall result in the structure `RobotControl_Request` defined below:

```
struct RobotControl_Request {
  RequestHeader header;
  RobotControl_Call request;
};
```

8.5.4.1.4 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. In these rules the token ${interfaceName} shall be replaced with the name of the interface:

1. The *Return* union name shall be "${interfaceName}_Return".
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 "unknown_operation" of type `UnknownOperation`.
4. The *Return* union shall have a case label for each operation in the interface:
    a. The integral value of the case label shall be the least significant 4 bytes of the MD5 hash applied to the operation name.

b. The member name for the case label shall be the operation name and the type shall be ${interfaceName}_${operationName}_Result, as defined in section 8.5.4.1.2

For example, the `RobotControl` interface defined in section 8.3.1.3.1 shall result in the union `RobotControl_Return` defined below:

```
union RobotControl_Return switch(long) {
  default:   UnknownOperation unknown_operation;
  case HASH("start"):    RoboControl_start_Result start;
  case HASH("stop"):     RoboControl_stop_Result stop;
  case HASH("setSpeed"): RoboControl_setSpeed_Result setSpeed;
  case HASH("getSpeed"): RoboControl_getSpeed_Result getSpeed;
};
```

In the above IDL the short-hand HASH(a_string) is used for clarity. In the actual IDL it shall be substituted with the actual computed value.

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 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 `ReplyHeader.`
    b. The second member of the *Reply* type shall be named "reply" and be of type ${interfaceName}_Return.

For example, the `RobotControl` interface defined in section 8.3.1.3.1 shall result in the structure `RobotControl_Reply` defined below:

```
struct RobotControl_Reply {
  ReplyHeader header;
  RobotControl_Return reply;
};
```

8.5.4.1.5 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 (one for request and one for reply) for every interface in the hierarchy. As a result, a service implementing a hierarchy of N interfaces (including derived), shall have N request topics, N reply topics and will consequently necessitate N DataWriters and N DataReaders. The types and the topic names are obtained as specified above. The *Request* and *Reply* types for a derived interface includes operations defined only in the derived interface.

For example the Basic Service Mapping rules applied to the Calculator interface defined in 8.3.1.3.2 result in the following request and reply Topic types:

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

struct Adder_add_Out {
  long return_;
};

union Adder_add_Result switch (long) {
  default:  UnknownException unknown_exception;
  case 0: Adder_add_Out out;
  case 1: SystemExceptionCode_sysx_;
};

union Adder_Call switch (long) {
  default:     UnknownOperation  unknown_operation;
  case HASH("add"):   Adder_add_In add;
};


struct Adder_Request {
  RequestHeader header;
  Adder_Call request;
};

union Adder_Return switch(long) {
  default:   UnknownOperation unknown_operation;
  case HASH("add"):    Adder_add_Result add;
};

struct Adder_Reply {
  ReplyHeader header;
  Adder_Return reply;
};

struct Subtractor_sub_In {
  long a;
  long b;
};

struct Subtractor_sub_Out {
  long return_;
};
```

```
union Subtractor_sub_Result switch (long) {
  default:  UnknownException unknown_exception;
  case 0: Subtractor_sub_Out out;
  case 1: SystemExceptionCode_sysx_;
};
union Subtractor_Call switch (long) {
  default:   UnknownOperation  unknown_operation;
  case HASH("sub"):  Adder_sub_In sub;
};


struct Subtractor_Request {
    RequestHeader header;
    Subtractor_Call request;
};
union Subtractor_Return switch(long) {
  default:   UnknownOperation unknown_operation;
  case HASH("sub"):    Subtractor sub_Result sub;
};

struct Subtractor_Reply {
  ReplyHeader header;
  Subtractor_Return reply;
};


struct Calculator_on_In {
  long dummy;
};

struct Calculator_off_In {
  long dummy;
};

struct Calculator_on_Out {
  long dummy;
};
union Calculator_on_Result switch(long) {
  default:  UnknownException unknown_exception;
  case 0: Calculator_on_Out out;
  case 1: SystemExceptionCode_sysx_;
};


struct Calculator_off_Out {
  long dummy;
```

```
};
union Calculator_off_Result switch(long) {
  default:  UnknownException unknown_exception;
  case 0: Calculator_off_Out out;
  case 1: SystemExceptionCode_sysx_;
};


union Calculator_Call switch (long) {
  default:  UnknownOperation  unknown_operation;
  case HASH("on"):  Calculator_on_In on;
  case HASH("off"): Calculator_off_In off;
};
struct Calculator_Request {
  RequestHeader header;
  Calculator_Call request;
};


union Calculator_Return switch(long) {
  default:   UnknownOperation unknown_operation;
  case HASH("on"):  Calculator_on_Result on;
  case HASH("off"): Calculator_off_Result off;
};

struct Calculator_Reply {
  ReplyHeader header;
  Calculator_Return reply;
};
```

The service implementation of the `Calculator` service shall create three DDS DataReaders and three DDS DataWriters. The DataReaders shall subscribe to topics “Calculator_Request”, “Adder_Request”, and “Subtractor_Request”. The DataWriters shall publish to the topics “Calculator_Reply”, “Adder_Reply”, and “Subtractor_Reply”.

A client to the `Calculator` service shall create three DDS DataReaders and three DDS DataWriters. The DataReaders shall subscribe to topics “Calculator_Reply”, “Adder_Reply”, and “Subtractor_Reply”. The DataWriters shall publish to the topics “Calculator_Request”, “Adder_Request”, and “Subtractor_Request”.

8.5.4.2 Enhanced Service Mapping of Interfaces

The Enhanced Service Mapping uses the [DDS-XTypes] type system in addition to the annotations defined in Section 8.5.1.

8.5.4.2.1 Mapping of Operations to the Request Topic Types

The mapping for interface operations 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:

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 name 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 have the @Empty annotation and contain a single member of type long and name "dummy".
4. If the operation has the @Mutable annotation, then the *In* structure shall have the annotation @Extensibility(MUTABLE_EXTENSIBILITY)

For example, the operations in the `RobotControl` interface defined in section 8.3.1.3.1 shall map to the following *In* structures.

```
@Empty struct RobotControl_start_In { };

@Empty struct RobotControl_stop_In { };

struct RobotControl_setSpeed_In {
  long s;
};

@Empty struct RobotControl_getSpeed_In { };
```

8.5.4.2.2 Mapping of Operations to the Reply Topic Types

Each operation in the interface shall map to an *Out* structure and a *Result* structure.

The following rules define the *Out* structure. 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:

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. If the operation defines a return value the *Out* structure shall have a first member named "return_". The type of the "return_" member shall be the return type of the function. In

the case where the function does not define a return value this member shall not be present.

b. The *Out* structure shall one additional 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.

c. If the operation has no return value, no inout parameters and no out parameters, the *Out* structure shall be annotated with the @Empty annotation and contain a single member with name dummy and type long.

For example, the operations defined in the `RobotControl` interface defined in section 8.3.1.3.1 map to the following *Out* structures:

```
@Empty struct RobotControl_start_Out { long dummy; };

@Empty struct RobotControl_stop_Out { long dummy; };

struct RobotControl_setSpeed_Out {
  boolean return_;
  long s;
};

struct RobotControl_getSpeed_Out {
  long return_;
};
```

The following rules define the *Result* structure. 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:

1. The *Result* structure name shall be "${interfaceName}_${operationName}_Result".
2. The *Result* structure shall have the annotation @Choice.
3. The *Result* structure shall have the annotation @Autoid
4. The *Result* structure shall have a first member with name "out" and with type ${interfaceName}_${operationName}_Out.
5. The *Result* structure name shall have a second member with name "sysx_" and with type SystemException.
6. The *Result* structure name shall have additional members for each exception declared as a possible outcome of the operation.

   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.

For example, the operations defined in the `RobotControl` interface defined in section 8.3.1.3.1 map to the following *Result* structures:

```
@Choice @Autoid
struct RobotControl_start_Result {
   RobotControl_start_out out;
   SystemExceptionCode sysx_;
};

@Choice @Autoid
struct RobotControl_stop_Result {
    RobotControl_stop_Out out;
    SystemExceptionCode sysx_;
};

@Choice @Autoid
struct RobotControl_setSpeed_Result {
    RobotControl_setSpeed_Out out;
    SystemExceptionCode sysx_;
    TooFast toofast_ex;
};

@Choice @Autoid
struct RobotControl_getSpeed_Result {
    RobotControl_getSpeed_Out out;
    SystemExceptionCode sysx_;
};
```

8.5.4.2.3 Interface Mapping for the Request Types

Each interface shall map to a *Request* structure.

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

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.

For example, the `RobotService` interface defined in section 8.3.1.3.1 shall result in the `RobotControl_Request` structure defined below:

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

```
    RobotControl_start_In start;
    RobotControl_stop_In stop;
    RobotControl_setSpeed_In setSpeed;
    RobotControl_getSpeed_In getSpeed;
};
```

8.5.4.2.4 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.

For example, the `RobotService` interface defined in section 8.3.1.3.1 maps to the following structure.

```
@Choice @Autoid
struct RobotControl_Reply {
    RobotControl_start_Result start;
    RobotControl_stop_Result stop;
    RobotControl_setSpeed_Result setSpeed;
    RobotControl_getSpeed_Result getSpeed;
};
```

8.5.4.2.5 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*...

For example the Enhanced Service Mapping rules applied to the `Calculator` interface defined in 8.3.1.3.2 result in the following request and reply Topic types:

```
struct Adder_add_In {
  long a;
  long b;
```

```
};

struct Adder_add_Out {
  long return_;
};

@Choice @Autoid
struct Adder_add_Result {
  Adder_add_Out out;
  SystemExceptionCode_sysx_;
};



struct Subtractor_sub_In {
  long a;
  long b;
};

struct Subtractor_sub_Out {
  long return_;
};

@Choice @Autoid
struct Subtractor_sub_Result {
   Subtractor_sub_Out out;
   SystemExceptionCode_sysx_;
};

@Empty
struct Calculator_on_In {
  long dummy;
};

@Empty
struct Calculator_off_In {
  long dummy;
};

@Empty
struct Calculator_on_Out {
  long dummy;
};

struct Calculator_on_Result {
```

```
   Calculator_on_Out out;
   SystemExceptionCode_sysx_;
};

@Empty
struct Calculator_off_Out {
  long dummy;
};

struct Calculator_off_Result {
   Calculator_off_Out out;
   SystemExceptionCode_sysx_;
};

@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;
};
```

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. For example, 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.

Consider the following example for multiple interface inheritance.

Comment [GP4]: This example seems incomplete. It seems you were about to describe how you can use a base interface to call a derived interface and vice-versa but that is all missing.

Original IDL	Synthesized DDS Topic Types for the Enhanced Service Profile

`module Test {`
 	
`exception Boo {};`
 	
`interface A {`
`  long foo(in long a1_in)`
`  raises (Boo);`
`};`
	`module Test {`
	
	`@Autoid struct A_foo_In {`
	`  long a1_in;`
	`};`
	
	`@Autoid struct A_foo_Out {`
	`  long return_;`

```
interface B {
  long bar(in long b_in,
           out long b_out);
};

interface D : A, B {
  double zoo(inout d_inout);
};

};
```

```
};

@Choice @Autoid struct A_foo_Result {
  A_foo_Out out;
    SystemExceptionCode sysx_;
    Boo boo_ex;
};

@Choice @Autoid struct A_Request {
    A_foo_In foo;
};

@Choice @Autoid struct A_Reply {
    A_foo_Result foo;
};

@Autoid struct B_bar_In {
   long b_in;
};

@Autoid struct B_bar_Out {
    long return_;
    long b_out;
};

@Choice @Autoid struct B_bar_Result {
    struct B_bar_Out out;
    SystemExceptionCode sysx_;
};

@Choice @Autoid struct B_Request {
    B_bar_In bar;
};

@Choice @Autoid struct B_Reply {
    B_bar_Result bar;
};

struct @Autoid D_zoo_In {
    long d_inout;
};

struct @Autoid D_zoo_Out {
    double return_;
    long d_inout;
};

@Choice @Autoid struct D_zoo_Result {
    D_zoo_Out out;
    SystemExceptionCode sysx_;
};

@Choice @Autoid struct D_Request {
```

```
    A_foo_In foo;
    B_bar_In bar;
    D_zoo_In zoo;
};

@Choice @Autoid struct D_Reply {
    A_foo_Result foo;
    B_bar_Result bar;
    D_zoo_Result zoo;
};

};
```

8.6 Discovering and Matching RPC Services

8.6.1 Client and Service Discovery for the Basic Service Mapping

The Basic Service Mapping relies on the built-in discovery provided by the underlying DDS implementation. It does not require any extensions to the DDS discovery builtin Topics..

A service client comprises one (or more) DataWriters used to send requests and corresponding DataReaders to receive the replies. Information on these DataWriters and DataReaders is propagated via the building DDS discovery protocol and the service-side will discover them.

Similarly a service implementation (service-side) comprises one (or more) DataReaders used to receive requests and corresponding DataWriters to send the replies. Information on these DataWriters and DataReaders is propagated via the building DDS discovery protocol and the client-side will discover them.

Matching of the client-side DDS Entities with their corresponding service-side DDS Entities shall rely on the regular DDS matching rules regarding Topic name matching, data-types, and compatible QoS. The Basic Service Mapping makes no modification to the standard DDS Entity matching rules.

Non-normative note: Service discovery for the Basic Service Mapping is not robust in the sense that discovery race conditions can cause the first 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 a client DataWriter and the service DataReader to mutually discover each other before the client DataReader and service DataWriter mutually discover other. If this situation occurs any requests made by the client may lose the reply sent by the service. To avoid the race condition the relationship between the client DataWriter and DataReader entities needs to be known to the service side and similarly the client-side needs to also know the relationship between the service DataReader and DataWriter. Communicatind these relationships require extensions to the DDS discovery information which can only be addressed by the Enhanced Service Mapping.

Comment [GP5]: This is redundant wuth the note that appears later. Maybe it should be droped and that Note moved here as it is more preceise.

8.6.2 Client and Service Discovery for the Enhanced Service Mapping

The Enhanced Service Mapping also relies in the builtin-discovery service provided by the underlying DDS implementation. However it extends the builtin-Topic data propagated by DDS discovery to contain information that can be used to relate a DataWriter with a DataReader. This

information shall be used by the client and service layer to implement more robust service invocation free of discovery race conditions.

8.6.2.1 Extensions to the DDS Discovery Builtin Topics

The DDS standard [1] specifies the existence of the ***DCPSPublication*** and ***DCPSSubscription*** builtin Topic and a corresponding builtin data writers and readers to communicate these topics. These builtin Topics, writers and readers are used to discover application-created DataReader and DataWriter entities.

The data type associated with the ***DCPSPublication*** builtin Topic is `PublicationBuiltinTopicData`. The data type associated with the ***DCPSSubscription*** builtin Topic is `SubscriptionBuiltinTopicData`. Both are defined in Section 2.1.5 of the DDS specification.

The DDS Interoperability Wire Protocol standard [2] specifies the serialization format of `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData`. The format is what the DDS Interoperability Wire Protocol calls a ParameterList whereby each member is serialized using CDR but preceded in the stream by the serialization of a short ParameterID identifying the member, followed by another short containing the length of the serialized member, followed by the serialized member. See Section 8.3.5.9 of the DDS Interoperability Wire Protocol [2]. This serialization format allows the `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData` to be extended without breaking interoperability.

8.6.2.1.1 Extended PublicationBuiltinTopicData

This DDS-RPC specification adds new members to the `PublicationBuiltinTopicData`. 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	0x00a0
related_datareader_key	BuiltinTopicKey_t	PID_RELATED_ENTITY_GUID	0x00a1

Comment [GP6]: Consider using 0x0054 is is available and fits well with the rest of the PIDs

```
@extensibility(MUTABLE_EXTENSIBILITY)
struct PublicationBuiltinTopicDataExt : PublicationBuiltinTopicData {
    @ID(0x00a0) BuiltinTopicKey_t service_instance_name;
    @ID(0x00a1) BuiltinTopicKey_t related_datareader_key;
};
```

When not present the default value for the `service_instance_name` shall be_the empty string.

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 Section 9.3.1.5 of the RTPS specification.

- Any `PublicationBuiltinTopicData` that corresponds to a DDS `DataWriter` that is not associated with a DDS Service, that is, it is no the client-side `DataWriter` used to sent requests nor the service-side `DataWriter` used to send back the replies, shall have the `related_datareader_key` set to GUID_UNKNOWN.
- The `PublicationBuiltinTopicData` that corresponds to the client-side `DataWriter` used to send requests for an DDS-RPC Service shall have the `related_datareader_key` set to the `BuiltinTopicKey_t` of the client-side `DataReader` used to receive the replies sent to the DDS-RPC Service.
- The `PublicationBuiltinTopicData` that corresponds to the service-side `DataWritre` used to send replies for an DDS-RPC Service shall have the `related_datareader_key` set to the `BuiltinTopicKey_t` of the service-side `DataReader` used to receive the requests sent to the DDS-RPC Service.

8.6.2.1.2 Extended SubscriptionBuiltinTopicData

This DDS-RPC specification adds new members 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	0x00a0
related_datawriter_key	BuiltinTopicKey_t	PID_RELATED_ENTITY_GUID	0x00a1

Comment [GP7]: Consider using 0x0054 is is available and fits well with the rest of the PIDs

```
@extensibility(MUTABLE_EXTENSIBILITY)
struct SubscriptionBuiltinTopicDataExt : SubscriptionBuiltinTopicData {
    @ID(0x00a0) BuiltinTopicKey_t service_instance_name;
    @ID(0x00a1) BuiltinTopicKey_t related_datawriter_key;
};
```

When not present the default value for the `service_instance_name` shall be the empty string.

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

- When not present the default value for the `related datawriter_key` shall be interpreted as GUID_UNKNOWN as defined by Section 9.3.1.5 of the RTPS specification.
- Any `SubscriptionBuiltinTopicData` that corresponds to a DDS `DataReader` that is not associated with a DDS Service, that is, it is no the client-side `DataReader` used to

receive replies nor the service-side `DataReader` used to receive the requests, shall have the `related datawriter_key` set to GUID_UNKNOWN.

- The `SubscriptionBuiltinTopicData` that corresponds to the client-side `DataReader` used to receive replies for an DDS-RPC Service 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.

8.6.2.2 Enhanced algorithm for Service Discovery

As previously mentioned DDS-RPC client-side contains a `DataWriter` for sending requests and a `DataReader` to receive the replies. Similarly, the service-side contains a `DataReader` for receiving the requests and a `DataWriter` for sending the replies.

The builtin DDS discovery mechanism ensures that DataWriters eventually discover the matched DataReaders and vice versa. However to ensure robust operation Service Discovery must ensure that no replies are lost due to race-conditions and transient states in the discovery process of the underlying DDS entities. To achieve robust operation additional rules shall be applied to the client-side DataWriter and server-side DataReader:

- The client-side DataWriter (request DataWriter) shall not send out a request to a Service DataReader unless the client-side DataReader (reply DataReader) has discovered and matched the corresponding service reply DataWriter.
- The service-side DataReader (request DataReader) shall rejects a requests if the service-side DataWriter (reply DataWriter) has not discovered and matched the corresponding client reply DataReader.

These rules ensure that when the replies are sent by the service DataWriter, the replies are not dropped simply because the discovery (of reply Topic) is not yet complete.

The rules above may be implemented within the logic of the “service” layer defined by this specification. There are no discovery algorithm modifications required in the underlying DDS implementation to achieve “service discovery” beyond the extensions to the built-in topic data described in section 8.6.2.1.

8.6.2.2.1 Service-side Client Matching

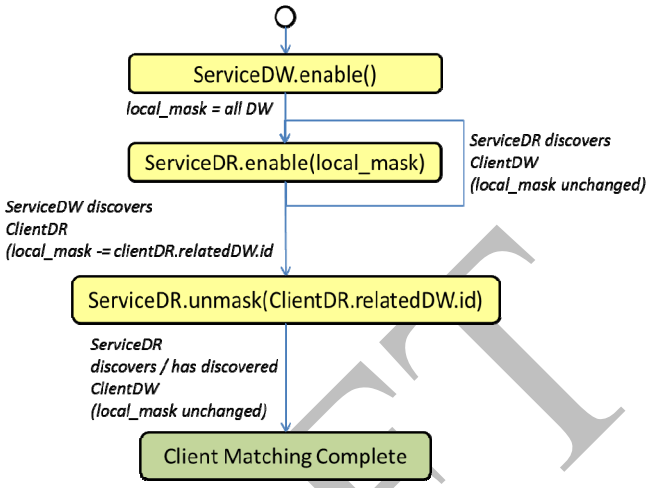

Figure 4: The service-side client matching algorithm

Figure 4 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 not complete the service-side does not process requests from the client. This behavior is illustrated in Figure 4 using a notional local state called `local_mask`. At the beginning, the service-side DataReader is enabled with `local_mask` so that it does not precess any the 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.

8.6.2.2.2 Cient-side Service Matching

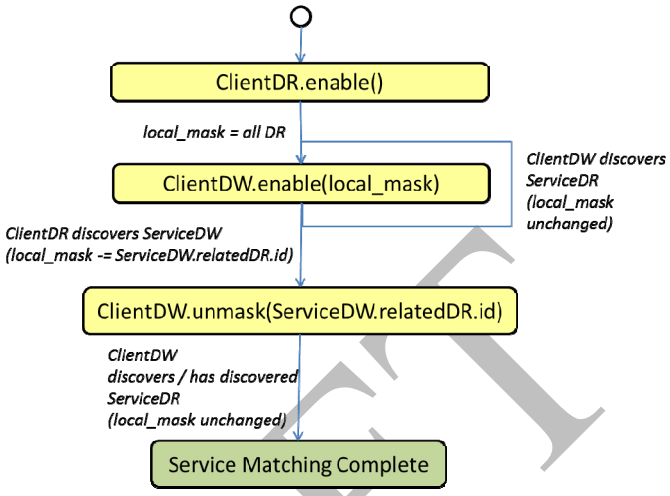

Figure 5: The client-side service matching algorithm

Figure 5 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 send requests to the service until "service matching" completes. This behavior is illustrated in Figure 5 using a notional local state called `local_mask`. At the beginning, the client-side DataWriter is enabled local_mask such that it does not sent any request at all. Attempts to send requests would be queued or fail When the client-side DataReader completes the discovery of a service DataWriter, the client DataWriter changes the local mask so that it can send requests as long as the service DataReader has been discovered.. The "service matching" algorithm completes when the client-side has discovered both the service end-points.Interface Evolution

8.6.3 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.

8.6.3.1 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, use of the unimplemented operation will always result in UNIMPLEMENTED system exception code.

8.6.3.2 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.

8.6.3.3 Changing Operation Signature

The Basic Service Profile does not support any modifications to the operation signatures.

8.6.4 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 the old interface and the service might be using the new interface or vice versa. Interface evolution is constrained by the rules described herein.

In subsections that follow, I_{old} represents the old interface whereas I_{new} represents the evolved interface. The interface mapping rules in the previous section 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}.

8.6.4.1 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 `SystemExceptionCode = NOT_IMPLEMENTED`. As the ids of the `_sysx` member match in $Reply_{old}$ and $Reply_{new}$, the sample is assignable and the C_{new} receives `NOT_IMPLEMENTED SystemExceptionCode`.

8.6.4.2 Removing an Operation

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

8.6.4.3 Reordering Operations and Base Interfaces

Reordering operations has no impact on the semantics of the interface. Theoretically, 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 they contain the same operations. Reordering base interfaces has no impact on the semantics of the interface.

8.6.4.4 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.]

8.6.4.5 Changing Operation Signature

8.6.4.5.1 Adding and Removing Parameters

Adding and removing parameters in an operation shall break compatibility with the older interface. The `@final` annotation on the *call* and *out* structures enforce this restriction. It is, however, possible to mark an operation `@mutable_operation` and `@not_optional_parameters` to allow the *call* and *out* structures to be not final.

8.6.4.5.2 Reordering Parameters

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

8.6.4.5.3 Changing the Type of a Parameters

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

8.6.4.5.4 Adding and Removing Return Type

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

8.6.4.5.5 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].

8.6.4.5.6 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 `UNEXPECTED_EXCEPTION` if the RPC call results into an exception that client does not understand.

8.7 Request and Reply Correlation

8.7.1 Request and Reply Correlation in the Basic Service Profile

The Basic service profile uses the RequestHeader and ReplyHeader described in Section 8.5.2.

8.7.2 Request and Reply Correlation in the Enhanced Service Profile

To support propagation of the sample identity of a related request this specification extends the DDS Interoperability Wire specification [REF] adding a new ParamaterID *PID_RELATED_SAMPLE_IDENTITY* with value 0x0072. This parameter may appear in the inlineQos submessage element of the DATA Submessage (see Section 9.4.5.3 of the DDS Interoperability Wire Protocol [2]).

The *PID_RELATED_SAMPLE_ID* is intended for the data samples that are sent as reply to a request message. When present is shall contain the CDR serialization of the `SampleIdentity_t` structure defined below. There is a special constant defined SAMPLE_IDENTITY_INVALID that is used to indicate an invalid sample identity.

When this feature is used the value of the *PID_RELATED_SAMPLE_ID* that appears in a reply message shall correspond to the sample identity (GUID, SN) of the request sample that triggered the reply.

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

RELATED_SAMPLE_ID_INVALID is defined as { GUID_UNKNOWN , SEQUENCENUMBER_UNKNOWN }. These constants are defined in the DDS Interoperability Wire Protocol specification [2].

8.7.2.1 Retrieve the Request Identity at the Service Side

This specification provides APIs to retrieve the sample identity of the request conveniently in the form of a `SampleIdentity_t` object.

8.7.2.2 Retrieve the Request Identity at the Client Side

This specification provides APIs to retrieve the correlation-id of the reply conveniently from the reply `SampleInfo` in the form of a `SampleIdentity_t` object. A `Sample` is an encapsulation of the data and `SampleInfo` as defined in [DDS-CXX-PSM] and [DDS-Java-PSM] platform specific mappings. This specification extends the `Sample` API to retrieve the related sample identity.

8.7.2.3 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 APIs to provide the related sample identity when sending the reply to a request.

8.8 Service Lifecycle

8.8.1 Activating Services with Basic Mapping

The Basic Service Profile provides interface-level control over which operations are activated when a service is activated. When used with inheritance hierarchy, the Basic Service Profiles supports activation of only a subset of interfaces. The Basic Service Profile allows service implementation of each interface in an hierarchy in a different address space. Such flexibility of activating only a subset of interfaces may be particularly attractive for the service implementation. The interfaces to be activated are passed as a regular expression when creating the service using either the RPCRuntime class or Requester/Service classes.

8.8.2 Activating Services with Enhanced Mapping

Creating an instance (subject to language binding) of the generic `Service` type implicitly activates all operations in the service. Service activation results in creation of a DataReader for requests and a DataWriter for replies. Service creation implicitly creates and enables the underlying entities. When the request/reply language binding is used, the `Service` is a generic type that is parameterized using the request type and reply type described in section 8.6.3. The topic name, if chosen, can be provided as one of the constructor parameters. The `RPCRuntime` class provides a portable way to initialize services when function-call style language binding is used.

8.8.3 Processing Requests

This specification provides three ways to receive 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 Service instance and the Service API can be used to retrieve the requests and send the replies. Unlike the synchronous service listener callback, the asynchronous service listener allows decoupling of request reception from request processing.
3. Independent of callbacks, the *Service* API, such as `waitForRequests` and `sendReply`, which can be used to retrieve the requests. See section 8.10.

8.8.4 Deactivating Services

Deleting (subject to language binding) the service instance deactivates the service. Deactivation implies deleting the underlying DDS Entities.

8.9 Service QoS

8.9.1 Interface/Operation Qos Annotation

To set the specific Qos for an operation, the IDL annotation extension presented in the [DDS-XTypes] specification shall be used. The built-in annotation would be:

```
@Annotation
local interface Qos
{
  attribute string RequestProfile;
  attribute string ReplyProfile;
};
```

The annotation can be applied at interface level. The profile attributes are string URLs of the form: `<protocol>://path/resource` where protocol and path are both optional. Support for the **Error! Hyperlink reference not valid.** protocol is mandatory. If the protocol is omitted, the string attributes are interpreted as if the protocol is **Error! Hyperlink reference not valid.**

The QoS profiles are XML QoS Profiles as defined in the [DDS-LWCCM] specification.

8.9.2 Default QoS

When no QoS is specified, default QoS are in effect. The default QoS for the endpoints (datareaders and datawriters) in both requester and service supports are specified as follows. The users may modify the QoS policies.

QoS Policy	Value
Reliability	DDS_RELIABLE_RELIABILITY_QOS
History	DDS_KEEP_ALL_HISTORY_QOS
Durability	DDS_VOLATILE_DURABILITY_QOS

8.10 Service API

8.10.1 Request-Reply API Style

8.10.1.1 Request-Reply API Style Platform Independent Model (PIM)

The platform independent model (PIM) for RPC over DDS consists of following entities.

8.10.1.1.1 Requester

The Requester provides a type-safe interface to call a remote procedure.

Requester
No attributes
Operations

new		Requester
	In: participant	DomainParticipant
	In: service_name	String
	In: request_topic_name	String
	In: reply_topic_name	String
	In: datareader_qos	DataReaderQoS
	In: datawriter_qos	DataWriterQoS
	In: publisher	Publisher
	In: subscriber	Subscriber
call		ReplyType
	In: request	RequestType
	In: timeout	Duration
call		ReplyType
	In: request	RequestType
	Out: reply_info	SampleInfo
	In: timeout	Duration
call_async		void
	In: request	RequestType
	Out: reply_future	Future
call_oneway		Void
	In: request	RequestType
get		ReplyType
	In: future	Future
wait		Void
	In: future	Future
	In: timeout	Duration
setReplyDataReaderQoS	In: drqos	DataReaderQoS
setRequestDataWriterQoS	In: dwqos	DataWriterQoS

bind		Boolean
	In: instance_name	String
unbind		Boolean
is_bound		Boolean
get_service_info		Void
	Out: service_name	String
	Out: instance_name	String

`Call` is a blocking method that invokes the remote service and waits for the reply. Two alternatives are provided depending upon sampleinfo is of interest or not. The `call_async` method invokes the remote service and returns immediately with a `future` object. The `future` object is used to retrieve the result of the invocation. `call_oneway` provides a way to make oneway calls. Oneway calls are guaranteed to execute at most once modulo failures. It is however not possible for the client to know if the invocation succeeded.

If the instance name of the service is not provided, the requests are sent to all the discovered service instances. The bind operation allows the requester to bind to a specific service instance if it has been already discovered. Once the service is bound, the requests are sent to the bound service instance only. The unbind operation removes the association.

8.10.1.1.2 Service

Service provides a type-safe interface to receive remote method invocations and send responses.

Service		
No attributes		
Operations		
new		Service
	In: participant	DomainParticipant
	In: service_name	String
	In: request_topic_name	String
	In: reply_topic_name	String
	In: datareader_qos	DataReaderQoS
	In: datawriter_qos	DataWriterQoS

	In: publisher	Publisher
	In: subscriber	Subscriber
receive		RequestType
	out: sample_info	SampleInfo
	out: identity	RequestIdentity
	In: timeout	Duration
wait		Boolean
	In: timeout	Duration
take_request		Boolean
	Out: request	RequestType
	Out: sample_info	SampleInfo
	Out: identity	RequestIdentity
read_request		RequestType
	Out: sample_info	SampleInfo
	Out: identity	RequestIdentity
reply		Void
	In: reply	ReturnType
	In: identity	RequestIdentity
setRequestDataReaderQoS	In: drqos	DataReaderQoS
setReplyDataWriterQoS	In: dwqos	DataWriterQoS

8.10.1.1.3 Service Listener

ServiceListener allows callback based synchronous service implementation. The listener is invoked when there is a request to be processed. However, the reply must be computed before the listener returns. In that sense it is synchronous. ServiceListener is abstract and implementations are required to implement the abstract methods. Note that request identity is provided as part of the callback.

ServiceListener (abstract)		
No attributes		
Operations		
process_request		ReplyType

	In: request	RequestType
	in: request_identity	RequestIdentity

8.10.1.1.4 Asynchronous Service Listener

ServiceListenerAsync allows callback based asynchronous service implementation. ServiceListenerAsync is abstract and implementations are required to implement the abstract methods. This listener is invoked when a request is available but the reply may be deferred. Note that request identity is not provided not as part of the callback. The request identity can be obtained using the Sample helper class. The request identity must be used when sending the reply.

ServiceListenerAsync (abstract)		
No attributes		
Operations		
on_request_available		Void
	in: service	Service

8.10.1.1.5 Reply Listener

ReplyListener allows callback-based notification to the requester that one or more replies are available. The listener is invoked when there is a reply to be processed. ReplyListener is abstract and implementations are required to implement the abstract methods. Note that request identity is provided in this callback API.

ReplyListener (abstract)		
No attributes		
Operations		
process_reply		ReplyType
	In: request	RequestType
	in: request_identity	RequestIdentity

8.10.1.1.6 Asynchronous Reply Listener

AsyncReplyListener allows callback based asynchronous notification of availability of one or more replies. AsyncReplyListener is abstract and implementations are required to implement the abstract methods. This listener is invoked when a reply is available. Note that request identity is not provided as part of the callback. The request identity can be obtained using the Sample helper class.

AsyncReplyListener (abstract)
No attributes

Operations		
on_reply_available		Void
	in: service	Service

8.10.1.1.7 Future

Future is a container (or proxy) for the result expected when the method invocation finishes sometime in the future. Future is intended to be used in combination with the Requester.

Future
No attributes
Operations

8.10.1.1.8 Synchronous, Asynchronous, and oneway Invocations

The Request/Reply style binding supports synchronous, asynchronous, and one-way invocations using `call`, `call_async`, and `call_oneway` operations defined in the Requester, respectively. The `call` operation is synchronous and blocks until the reply is available. The blocking duration is configurable. The `call_async` operation returns immediately with a `future`, which serves as a token for the reply. The `call_oneway` operation returns `void` and provides no future object because oneway operations always return `void`. It is possible to use `call` and `call_async` operations to invoke a oneway operation but in such cases the behavior is undefined.

8.10.1.2 Request-Reply Style Language Binding for C++

The types in the PIM map to C++ templates parameterized over the *request* and the *reply* types. The data and the `sampleinfo` are aggregated in a parameterized `Sample` type as in [DDS-CPP-PSM]. [Non-normative Note: The following section is included for readability and may have deviations from the normative reference [DDS-RPC-CXX].]

```
template <class TReq, class TRep>
class Requester {
public:

  // Creates a Requester with the minimum set of parameters.
  Requester (DomainParticipant *participant, const std::string &service_name);

  // Creates a Requester with parameters.
  Requester (const RequesterParams &params);

  void call(TRep &, const TReq &, const Duration &);

  void call(Sample<TRep> &, const TReq &, const Duration &);

  LoanedSamples<TRep> call(const TReq &, const Duration &);

  dds::future<Sample<TRep>> call_async(const TReq &);

  void call_oneway(const TReq &);
```

```
  void setRequestDataWriterQos(const DataWriterQos & dwqos);

  void setReplyDataReaderQos(const DataReaderQos & drqos);

  void wait_for_replier(unsigned howmany, const Duration &);

  bool bind(const std::string & instance_name);

  bool unbind();

  bool is_bound();

  void get_service_info(std::string & service_name, std::string & bound_instance);

};


template <class TReq, class TRep>
class Service {

public:

  // Creates a Service with the minimum set of parameters.
  Service (DomainParticipant *participant, const std::string &service_name);

  // Creates a Service with parameters.
  Service(const ServiceParams &params);

  // blocking take
  void receive(TReq &, RequestIdentity &, Duration);

  // blocking take with sampleinfo
  void receive(Sample<TReq> &, RequestIdentity &, Duration);

  // blocking take
  // SampleInfo will contain the request identity.
  LoanedSamples<TReq> receive(Duration);

  bool wait(Duration);

  // non-blocking take
  bool take_request(TReq &, RequestIdentity &);

  // non-blocking take with sampleinfo
  bool take_request(Sample<TReq> &, RequestIdentity &);

  // read-blocking take
  bool read_request(TReq &, RequestIdentity &);

  // non-blocking take with sampleinfo
  bool read_request(Sample<TReq> &, RequestIdentity &);

  bool reply(const TRep &, const RequestIdentity &);

  void setReplyDataWriterQos(const DataWriterQos & dwqos);

  void setRequestDataReaderQos(const DataReaderQos & drqos);

};


template <class TReq, class TRep>
class ServiceListener {
public:
  virtual TRep * process_request(const Sample<TReq> &, const RequestIdentity &) = 0;
  ~ServiceListener();
};
```

```
template <class TReq, class TRep>
class AsyncServiceListener {
public:

  virtual void on_request_available(Service<TReq, TRep> &) = 0;
  ~AsyncServiceListener();

};

template <class TRep>
class ReplyListener {
public:

  virtual void process_reply(const Sample<TRep> &, const RequestIdentity &) = 0;
  ~ReplyListener();
};

template <class TReq, class TRep>
class AsyncReplyListener {
public:

  virtual void on_reply_available(Requester<TReq, TRep> &) = 0;
  ~AsyncReplyListener();
};

class RequesterParams {
public:

    RequesterParams (DomainParticipant *participant);

    template <class TRep>
    RequesterParams &      reply_listener(ReplyListener<TRep> &listener);

    template <class TReq, class TRep>
    RequesterParams &      async_reply_listener(AsyncReplyListener<TReq, TRep> &listener);

    RequesterParams &      service_name (const std::string &name);

    RequesterParams &      request_topic_name (const std::string &name);

    RequesterParams &      reply_topic_name (const std::string &name);

    RequesterParams &      datawriter_qos (const DataWriterQos &qos);

    RequesterParams &      datareader_qos (const DataReaderQos &qos);

    RequesterParams &      publisher (Publisher *publisher);

    RequesterParams &      subscriber (Subscriber *subscriber);

};


class ServiceParams {
public:

    ServiceParams (DomainParticipant *participant)

    template <class TReq, class TRep>
    ServiceParams &        service_listener (ServiceListener<TReq, TRep> &listener);

    template <class TReq, class TRep>
    ServiceParams &        async_service_listener (AsyncServiceListener<TReq,TRep> &);

    ServiceParams &        service_name (const std::string &service_name);
```

```
    ServiceParams &        request_topic_name (const std::string &req_topic);

    ServiceParams &        reply_topic_name (const std::string &rep_topic);

    ServiceParams &        datawriter_qos (const DataWriterQos &qos);

    ServiceParams &        datareader_qos (const DataReaderQos &qos);

    ServiceParams &        publisher (Publisher *publisher);

    ServiceParams &        subscriber (Subscriber *subscriber);

};


template <class T>
class future {

  // will mimic std::future in C++11 to the extent possible in C++03.

public:

    T get();
    void wait();
    void wait(Duration);
};
```

8.10.1.3 Request-Reply Style Language Binding for Java 5

The types in the PIM map to generic Java classes parameterized over the *request* and the *reply* types. The data and the `sampleinfo` are aggregated in a parameterized `Sample` type as in [DDS-Java-PSM]. [Non-normative Note: The following section is included for readability and may have deviations from the normative reference [ddsrpc4j].]

```
package org.omg.dds.rpc;

import org.omg.dds.domain;
import org.omg.dds.pub;
import org.omg.dds.sub;
import org.omg.dds.pub;

public class Requester<TReq, TRep> {

  // Creates a Requester with the minimum set of parameters.
  public Requester (DomainParticipant participant, String service_name);

  // Creates a Requester with parameters.
  public Requester (RequesterParams params);

  public void call(TRep reply, TReq request, Duration duration);

  public void call(Sample<TRep> sample, TReq request, Duration duration);

  public LoanedSamples<TRep> call(TReq request, Duration duration);

  public java.util.concurrent.future<Sample<TRep>> call_async(TReq request);

  void setRequestDataWriterQos(DataWriterQos dwqos);

  void setReplyDataReaderQos(DataReaderQos drqos);
```

```
};


public class Service <TReq, TRep> {

  // Creates a Service with the minimum set of parameters.
  public Service (DomainParticipant participant, String service_name);

  // Creates a Service with parameters.
  public Service(ServiceParams params);

  // blocking take
  public void receive(TReq request, RequestIdentity identity, Duration duration);

  // blocking take with sampleinfo
  public void receive(Sample<TReq> sample, RequestIdentity identity, Duration duration);

  // blocking take
  // SampleInfo will contain the request identity.
  public LoanedSamples<TReq> receive(Duration duration);

  public boolean wait(Duration duration);

  // non-blocking take
  public boolean take_request(TReq request, RequestIdentity identity);

  // non-blocking take with sampleinfo
  public boolean take_request(Sample<TReq> sample, RequestIdentity identity);

  // read-blocking take
  public boolean read_request(TReq request, RequestIdentity identity);

  // non-blocking take with sampleinfo
  public boolean read_request(Sample<TReq> sample, RequestIdentity identity);

  public boolean reply(TRep reply, RequestIdentity identity);

  void setReplyDataWriterQos(DataWriterQos dwqos);

  void setRequestDataReaderQos(DataReaderQos drqos);

};


public interface ServiceListener< TReq, TRep> {

  TRep process_request(Sample<TReq> sample, RequestIdentity identity);

};


public interface AsyncServiceListener<TReq, TRep> {

  void process_request(Service<TReq, TRep> service);

};

public interface AsyncReplyListener <TReq, TRep> {

  void on_reply_available (Requester<TReq, TRep> requester);

}

public interface ReplyListener <TRep> {
```

```
  void process_reply(Sample<TRep> reply, RequestIdentity identity);
}
public class RequesterParams {
    public RequesterParams (DomainParticipant participant);
    public RequesterParams service_name (String name);
    public RequesterParams request_topic_name (String name);
    public RequesterParams reply_topic_name (String name);
    public RequesterParams datawriter_qos (DataWriterQos qos);
    public RequesterParams datareader_qos (DataReaderQos qos);
    public RequesterParams publisher (Publisher publisher);
    public RequesterParams subscriber (Subscriber subscriber);
};
public class ServiceParams {
    public ServiceParams (DomainParticipant participant)
    public <TReq, TRep> ServiceParams service_listener
                        (ServiceListener< TReq, TRep > listener);

    public <TReq, TRep> ServiceParams async_service_listener
                        (AsyncServiceListener< TReq, TRep > listener);
    public ServiceParams service_name (String service_name);
    public ServiceParams request_topic_name (String req_topic);
    public ServiceParams reply_topic_name (String rep_topic);
    public ServiceParams datawriter_qos (DataWriterQos qos);
    public ServiceParams datareader_qos (DataReaderQos qos);
    public ServiceParams publisher (Publisher publisher);
    public ServiceParams subscriber (Subscriber subscriber);
};
```

8.10.2 Function-Call API Style

8.10.2.1 Function-Call API Style Platform Independent Model (PIM)

This section defines the API used to connect to a server, register services on the server, and create proxies for use at the client-side.

This section defines the API used to register services, lookup services, and create proxies for use at the client-side. The API are described in a platform independent IDL. The language mapping for C++ and Java should be derived using either (1) the [IDL2C++11] mapping using only the language and library capabilities available in ISO IEC C++ 2003 language specification and (2) the [IDL2Java] specification for mapping IDL to Java.

8.10.2.1.1 Service Implementation

A DDS RPC Services declaration maps to an interface (Java Interface, an abstract class in C++) that must be implemented by a user-defined concrete class. This concrete class is referred to as a *servant*. The following base servant API is platform independent.

ServantBase		
No attributes		
Operations		
get_domainParticipant		DomainParicipant
get_serviceName		String
get_instanceName		String
get_request_datareader_qos		DataReaderQos
get_reply_datawriter_qos		DataWriterQos

ServantBase is inherited by the concrete implementation of a Service.

8.10.2.1.2 Service Registration and Resolution

The following interface defines the operations to register, and resolve a DDS RPC Service.

RPCRuntime		
No attributes		
Operations		
New		RPCRuntime
	In: domainParticipant	DomainParticipant
create_service		Void
	In: serviceImpl	ServantBase
	In: serviceName (optional)	String
	In: InstanceName (optional)	String
	In: drQos (optional)	DataReaderQos
	In: dwQos (optional)	DataWriterQos
create_client		ServiceProxy

	In: serviceName	String
	In: instanceName (optional)	String
start_service_processing		Void
	In: process_events_in_main	boolean

In most cases, serviceName is the same as the name of the interface but it need not be. The instanceName is user-defined. In case of name collision, the resolution is implementation-defined.

8.10.2.1.3 Using a Service

ServiceProxy represents the base interface of all the client-side handles for service invocation.

ServiceProxy		
No attributes		
Operations		
isBound		Boolean
get_service_info		Void
	Out: serviceName	String
	Out: instanceName	String
get_service_name		String
Bind		Void
	In: instance_name	String

The above operations are inherited by all the concrete `ServiceProxy` instances. The `isBound` operation returns true if the `ServiceProxy` object refers to a specific instance of a service. In addition to the general-purpose operations above, `ServiceProxy` is inherited by interface-specific proxy classes that implement the operations. Each operation has a synchronous and asynchronous variant. At least one of them must be provided. Furthermore, the synchronous variant of each operation takes an optional timeout as the last parameter to the operation. If the reply is not available before the timeout expires, TIMEOUT_EXCEPTION is raised to indicate the error condition.

8.10.2.1.4 Synchronous, Asynchronous, and oneway Invocations

For each operation in the interface, the proxy will contain a sync and async version of the procedure with the corresponding operation parameters. The async version needs also an extra parameter: a CallBackHandler object. The CallbackHandler includes three operations: on_registered, on_complete, and on_exception.

The on_request_sent operation is called when the request is sent. The middleware provides the request identity. It allows a single callback object to be used for multiple asynchronous requests. Each request has a unique request identity.

The On_complete operation is invoked when a reply is available for a previously sent request. This callback may be called more than once if there are more than one replies. Similarly, on_exception callback is called when async operation fails and throws a remote exception. This operation may also be called more than once depending upon how many Services reply. It is also possible that both on_complete and on_exception are invoked for the same request executed by two or more different service instances.

CallbackHandler		
No attributes		
Operations		
on_complete		Void
	in: identity	RequestIdentity
	Function return value and InOut/Out function parameters	Type as defined by the language binding
on_exception		Void
	In: exception	dds.rpc.Exception
	in: identity	RequestIdentity
on_request_sent		Void
	In: identity	RequestIdentity

8.10.2.2 Language Bindings for the Function-Call Style RPC over DDS

Readers are directed to [DDS-RPC-CXX] and [ddsrpc4j] (normative references) for further information. The following subsection is included as convenience to the readers.

Comment [GP8]: This does not seem enough for a specification. It should state how this specifications are used here. I.e. the intergface shall be mapped to…
Also these specifications are not saying anything about the RPCRuntime…

Need to define LanguageBindig (not just example) for C++ and Java.

8.10.2.2.1 Non-Normative Example

This section describes a non-normative example showing the function-call style language bindings for a Bank Account service.

8.10.2.2.2 Service Declaration

A Bank Account service is declared using IDL.

```
module Bank {

struct Address {
  String line1;
```

```
  String line2;
}
exception NotEnoughFunds {}

@DDSService
interface Account
{
  double  balance();
  boolean deposit(double amount);
  boolean withdraw(double amount) raises (NotEnoughFunds);
}

@DDSService
interface SavingsAccount : Account
{
  double interest_rate();
}

@DDSService
interface CheckingAccount : Account
{
  void order_checks(long howmany, Address mail);
}

}
```

8.10.2.2.3 C++ Implementation

8.10.2.2.3.1 C++ Interfaces (Infrastructure)

```
namespace dds { namespace rpc {

class ServantBase {
public:
  DomainParticipant get_domain_participant() const;
  std::string get_service_name() const;
  std::string get_instance_name() const;
  DataReaderQos get_request_datareader_qos() const;
  DataWriterQos get_reply_datawriter_qos() const;
};

class RPCRuntime
{
public:
  RPCRuntime(DomainParticipant * dp);

  void start_service_processing(bool process_events_in_main = false);
};

class Server
{
public:
   Server(const RPCRuntime &);

   void add_service(ServantBase &servant, std::string instance_name);
};

class ServiceProxyBase
{
public:
  ServiceProxyBase(const RPCRuntime &);
```

```
};

} // namespace rpc
} // namespace dds
```

8.10.2.2.3.2Generated C++ Interfaces

```
namespace Bank {
struct Address {
  std::string line1; // notional
  std::string line2; // notional
};

// Generated by IDLGEN from Bank.idl
class Account {
public:
  virtual double balance()=0;
  virtual bool deposit(double amount)=0;
  virtual bool withdraw(double amount)=0;
};


// Generated by IDLGEN from Bank.idl
class SavingsAccount : public virtual Account {
public:
  virtual double interest_rate()=0;
};

// Generated by IDLGEN from Bank.idl
class CheckingAccount : public virtual Account {
public:
  virtual void order_checks(int howmany, const Address &)=0;
};

class AccountProxy : public ServiceProxyBase
{
public:
  explicit AccountProxy(const RPCRuntime &runtime);

  double balance();

  bool deposit(double amount);

  bool withdraw(double amount);
};

class SavingsAccountProxy : public AccountProxy
{
public:

  explicit SavingsAccountProxy(const RPCRuntime &runtime);

  double interest_rate();
};

class CheckingAccountProxy : public AccountProxy
{
public:

  explicit CheckingAccountProxy(const RPCRuntime &runtime);

  void order_checks(int howmany, const Bank::Address &);
```

```
};


// Generated by IDLGEN from Bank.idl
class Bank::AccountServant
  : public virtual Bank::Account,
    public virtual ServantBase
{
public:

  virtual double  balance() {
   // empty implementation body
  }

  virtual bool deposit(double amount) {
   // empty implementation body
  }

  virtual bool withdraw(double amount) {
    // empty implementation body
  }
};

class SavingsAccountServant
  : public virtual Bank::SavingsAccount,
    public virtual ServantBase
{
public:

  virtual double interest_rate() {
    // empty implementation body
  }
};

template <>
class CheckingAccountServant
  : public virtual Bank::CheckingAccount,
    public virtual ServantBase
{
public:

  virtual void order_checks() {
    // empty implementation body
  }
};

} // namespace Bank
```

8.10.2.2.3.3C++ Service Implementation

```
namespace mybank {

// Service implemenatation
class MyCheckingAccountImpl : Bank::CheckingAccountServant
{
public:
  virtual double  balance() {
    // my implementation body
  }
  virtual bool deposit(double amount) {
    // my implementation body
  }
  virtual bool withdraw(double amount) {
    // my implementation body
```

```
  }
    virtual void order_checks() {
    // my implementation body
  }
};

} // namespace mybank

void create_service()
{
  // Creating a service
  RPCRuntime runtime;
  Server server = Server(runtime, "CheckingAccount");
  CheckingAccountServant servant("My1234CheckingAccount");
  server.add_service(servant);
  server.serve();
}
```

8.10.2.2.3.4Client Implementation in C++

```
void create_polymorphic_client()
{
  RPCRuntime runtime;
  Bank::CheckingAccountProxy ca_proxy(runtime, "CheckingAccount");

  ca_proxy.deposit(100);

  ca_proxy.order_checks(50, some_address);
}
```

9 RFP Requirements

The table below describes how this submission addresses the RFP requirements.

RFP Requirements	Section
6.5.1 Submissions shall define the syntax and types that can be used for defining a DDS Service. At minimum the specification shall support the declaration of DDS Services in IDL and Java.	See Section 8.3 for IDL See Section 8.3.2 for Java
6.5.2 Submissions shall allow the definition of synchronous, one-way, and asynchronous invocations.	See Sections 8.10.1.1.8 for request-reply style language binding and 8.10.2.1.4 for function-call style language binding.
6.5.3 Submissions shall allow DDS Quality of Service (QoS) to be associated with Services at an interface level. Submissions shall define how operations requests, replies and exceptions map to DDS Topics. The specification shall precisely define the exceptions to be raised under failure of local or remote operations.	See Section 8.9.1 for QoS association See Section 8.2.3 and 8.5 for mapping of operations to DDS topic types. See Section 8.5.3 for system and user-defined exceptions

6.5.4 Submissions shall define how services are discovered.	See Section 8.6.
6.5.5 Submissions shall define the language mapping for a DDS Service for at least the Java 5 and ISO C++ PSM.	For C++ See Section 8.10. Additionally, see [DDS-RPC-CXX] and [ddsrpc4j] normative references.
6.5.6 Submissions shall define an API to deal with service activation and deactivation.	See section 8.7
6.5.7 Submissions shall ensure that the proposed solution is interoperable between and portable across DDS implementations.	The Basic Service Profile has been designed to be portable across DDS implementations. Further, this submission is based on other standards such as [DDS-XTypes] and [DDS-RTPS]. Both profiles depend on the [DDS-RTPS] specification. Some enhancements proposed in this specification use the extensibility mechanisms supported by the above mentioned standards. (e.g., `relateSampleId` as inline-QoS and `relatedEntity` as built-in topic data, which is mutable)
6.5.8 Submissions shall avoid any RPC-specific extension to the DDSI/RTPS protocol and provide a solution that works at a DCPS level.	The Base Service Profile avoids any RPC-specific extensions to the DDSI/RTPS protocol and works at DCPS level.
6.5.9 Submissions shall ensure that DDS applications using DCPS are able to publish/subscribe topics associated to service operations and as a result be capable of generating service request/reply.	This specification complies with this requirement as the services use DDS topics for requests and replies. To this end, this specification is fully data-centric and therefore interoperability can be achieved easily as long as a shared type description (IDL, XSD, Java Interfaces, etc.) exists.
Optional Requirements	**Section**
6.6.1 Submissions may define a mapping from a subset of legal CORBA Interfaces to RPC over DDS .	Not supported.
6.6.2 Submissions may allow QoS to be associated with Services at and operation level.	Both service profiles supports QoS to be associated at Service as well as at the Operation granularity.
6.6.3 Submissions may define an API to perform map/reduce operations.	Not supported.
6.6.4 Submissions may allow QoS changes at a	See set*Qos operations in the Requester and

service instance level, for those QoS Policies that are mutable.	Service classes. Only mutable Qos can be changed through this API.

References

[1] DDS: Data-Distribution Service for Real-Time Systems version 1.4.
http://www.omg.org/spec/DDS/

[2] DDS-RTPS: Data-Distribution Service Interoperability Wire Protocol version 2.2,
http://www.omg.org/spec/DDS-RTPS/

[3] DDS-XTYPES: Extensible and Dynamic Topic-Types for DDS version 1.0
http://www.omg.org/spec/DDS-XTypes/

[4] OMG-IDL: Interface Definition Language (IDL) version 3.5,
http://www.omg.org/spec/IDL35/