Date: September 2024

OBJECT MANAGEMENT GROUP

DDS Status Monitoring

Version 1.0 – beta 1

OMG Document Number: dtc/24-09-15

Normative Reference: http://www.omg.org/spec/DDS-Monitoring/1.0

Associated Normative Machine Consumable Files:

http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_dds_distribution.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_dds_distribution_constants.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_dds_common.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_dds_status.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_dds_qos.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_dds_entities.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_annotations.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_statistics.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_logging.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_resource.idl
http://www.omg.org/spec/DDS-Monitoring/20240801/monitoring_administration.idl

---

This OMG document replaces the submission document (c4i/24-08-01). It is an OMG Adopted Beta Specification and is currently in the finalization phase. Comments on the content of this document are welcome and should be directed to issues@omg.org by December 16, 2024.

You may view the pending issues for this specification from the OMG revision issues web page https://issues.omg.org/issues/lists.

The FTF Recommendation and Report for this specification will be published in September 2025. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

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://issues.omg.org/issues/create-new-issue).

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 Profiles

Modernization Specifications

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

- CORBAServices
- CORBAFacilities

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

Signal and Image Processing Specifications

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/Liberation Serif – 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 via the report form at:

http://issues.omg.org/issues/create-new-issue

This page intentionally left blank.

1 Scope

The DDS Monitoring specification provides a framework for monitoring the health and performance of the DDS platform in live operational environments. This standard addresses the critical need for a unified and standardized approach to remotely observe the operational status of DDS systems, detect anomalies or degradations, and identify their root cause.

The specification defines the various elements needed to remotely observe and assess the operational health of a DDS System. This is broken into 3 separate aspects:

- The DDS Monitoring Resource Model. This describes the set of Resources contained in a DDS platform and the Observable Metrics associated with each Resource.
- The DDS Monitoring Distribution Model. This describes how the set of resources and associated observable elements can be accessed from a remote application over the network.
- The DDS Monitoring Administration API. This describes the API/protocol remote applications and tools can use to control the data collected as well as query current values of the Observable Elements.

2 Conformance

The Conformance clause identifies which clauses of the specification are mandatory (or conditionally mandatory) and which are optional for an implementation to claim conformance to the specification.

There is a single conformance profile for this speciation. To conform with the specification an implementation must implement the data model, distribution model, administration API, and protocols specified in clauses 7 and 8.

3 Normative References

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

[DDS] Object Management Group, Data Distribution Service, Version 1.4, https://www.omg.org/spec/DDS/1.4

[DDSI-RTPS] Object Management Group, Real-Time Publish-Subscribe (RTPS): The DDS-Interoperability Wire Protocol, Version 1.3, https://www.omg.org/spec/DDSI-RTPS/2.5

[DDS-XTYPES] Object Management Group, Extensible Types for DDS, Version 1.3, https://www.omg.org/spec/DDS-XTYPES/1.3

[DDS-RPC] Object Management Group, RPC over DDS, Version 1.0, https://www.omg.org/spec/DDS-RPC/1.0

[DDS-Security] Object Management Group, DDS Security, Version 1.1, https://www.omg.org/spec/DDS-Security/1.2

[IDL] Object Management Group, Interface Definition Language, Version 4.2, https://www.omg.org/spec/IDL/4.2

[POSIX fnmatch] The Open Group Base Specifications Issue 7, 2018 fnmatch function.
https://pubs.opengroup.org/onlinepubs/9699919799/functions/fnmatch.html

4 Terms and Definitions

For this specification, the following terms and definitions apply.

Attribute

Key-value pairs that provide additional contextual information about monitoring data. They can be associated with metrics, logs, and resources. Attributes allow for the annotation of monitoring data with rich, structured contextual information, making the data more meaningful and easier to analyze.

DDS

Data Distribution Service (DDS) is a family of standards from the Object Management Group (OMG, http://www.omg.org) that provide connectivity, interoperability, and portability for Industrial Internet, cyber-physical, and mission-critical applications. The DDS connectivity standards cover Publish-Subscribe (DDS), Service Invocation (DDS-RPC), Interoperability (DDSI-RTPS), Information Modeling (DDS-XTYPES), Security (DDS-Security), as well as programming APIs for C, C++, Java and other languages.

DDS Domain

Represents a global data space. It is a logical scope (or “address space”) for Topic and Type definitions. Each Domain is uniquely identified by an integer Domain ID. Domains are completely independent of each other. For two DDS applications to communicate with each other they must join the same DDS Domain.

Log

A (structured), unit of text produced by a system to inform of the occurrence of an event. E.g. a line of text indicating that a remote DDS Participant has been discovered.

Metric

A measured, numeric value representing one aspect of the state or performance of a system under observation. E.g. a counter indicating how many messages a DDS DataReader has in its cache.

Monitoring

The task of assessing the health of a system by collecting and analyzing Metrics, Logs, and Traces produces by the system.

Monitoring Data

A type of data that is not used as part of the application function but rather serves the purposes of observing application status and health. This data largely consists of numeric Metrics that are collected either periodically or whenever there is a significant change. Monitoring data may also contain non-numeric information representing configuration information, event occurrence, logs, and traces.

Monitoring Distribution Model

The data model and communication protocol or mechanism used to send the monitoring data values to external applications that collect, store, analyze, or visualize the data.

Monitoring Resource Model

The set of monitoring-relevant Resources associated with a specific software stack (e.g. the DDS Platform) and the Monitoring Data associated with each of those.

Observability

The extent to which the internal states of a system can be inferred from externally available data. An observable software system provides the ability to understand its operational behavior and detect issues, and their causality.

Resource

Resource is an abstraction that represents an application, platform, or middleware entity that can be monitored in the system. Resources have an identity and a lifecycle. Resources are referenceable and visible to the Monitoring Infrastructure. They provide context for the Monitoring Data In a DDS System the DDS Entities (DomainParcitpant, DataWriter, etc.) could be considered resources.

Trace

Structured data representing represents the entire journey of an individual action as it moves through all the nodes of a distributed system. E.g. the journey of a data message from the moment an application writes it using a DDS DataWriter to the moment another application receives it using a DDS DataReader.

5 Symbols

The following acronyms are used in this specification.

Table 5.1: Acronyms

Acronyms	Meaning
API	Application Programming Interface
DCPS	Data-Centric Publish-Subscribe
DDS	Data Distribution Service
OMG	Object Management Group
QoS	Quality-of-Service
RPC	Remote Procedure Call
RTPS	Real-Time Publish-Subscribe Protocol

6 Additional Information

6.1 Acknowledgments

The following individuals and companies contributed to this specification:

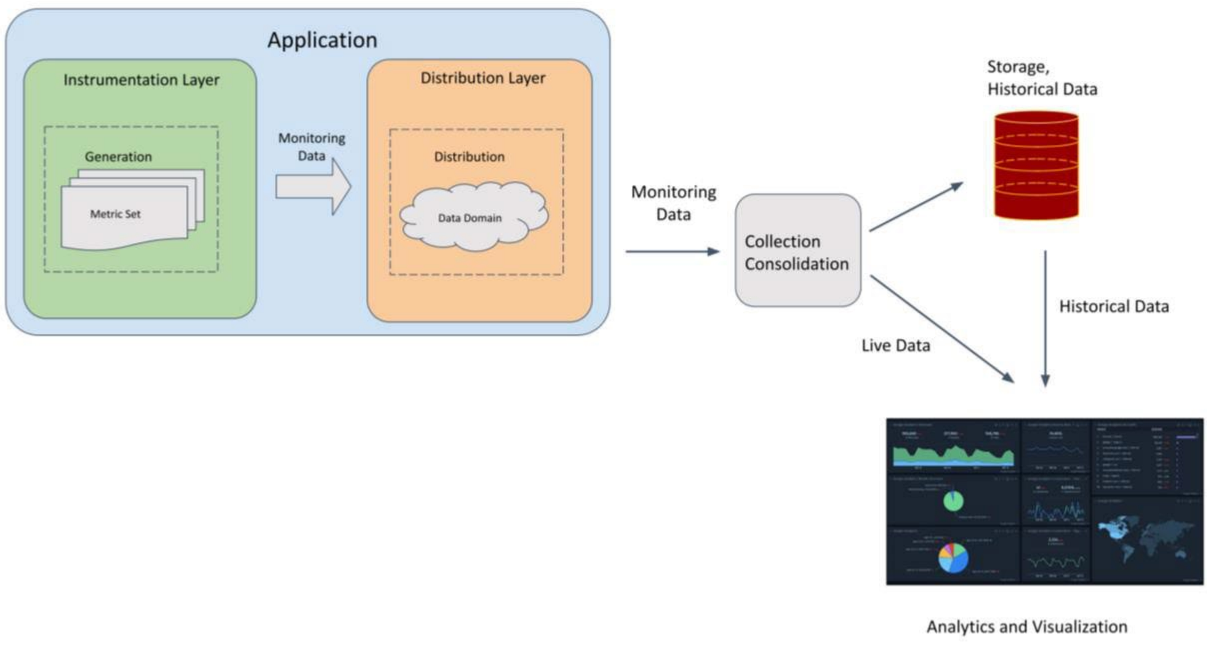
- Gerardo Pardo-Castellote, Real-Time Innovations, Inc.
- Matt Wilson, SimVentions, Inc.

7 Conceptual Model

7.1 Introduction

Monitoring the health status of a DDS System requires an Instrumentation Layer and a Distribution Layer:

- The **Instrumentation Layer** gathers Monitoring Data on the internal state of the DDS platform and the occurrence of relevant events (including logs and traces).
- The **Distribution Layer** makes the Monitoring Data available to other applications (and tools) that collect, store, analyze, and visualize the data to assess the health of the overall system and each of its components.



Def

Figure 8.1: Instrumentation and Distribution Layers

7.2 Concepts

Monitoring Data is a type of data that is not used as part of the application function, instead, it is used to observe application status and health. This data may consist of numeric values (metrics), non-numeric values (attributes), configuration information, event notifications, logs, and traces. The metric data may be generated periodically or when there is a significant change.

Monitoring Data must be associated with an application/user-relevant object (or entity) that provides the context for that information. For example, the Metric “CPU utilization” is associated with a specific Resource (e,g, an Operating System Process or a Hardware CPU core).

R*esources* are those identifiable objects/entities whose state and events can be observed, generating and providing the context for the monitoring data.

Resources may belong to different software (or hardware) layers, such as the underlying computing platform, the operating system, the middleware layers, or the application itself.

The *Monitoring Resource Model* is the set of monitoring-relevant Resources associated with a specific software stack (e.g. the DDS Platform) and the observable elements belonging to each resource.

The *Monitoring Distribution Model* is the communication protocol and data model used to deliver the Monitoring Data to other applications that store, process, analyze, and visualize the data.

- The DDS-Monitoring specification uses the OMG IDL4 language, extended using annotations defined in this specification, to define the *Monitoring Resource Model*. Likewise, it also uses IDL4 to define the data model used by the *Monitoring Distribution Model*.
- The DDS-Monitoring specification uses DDS, extended with additional built-in Topics included in the specification, as the communication protocol to distribute the Monitoring Data.

Note that the DDS domain (and middleware implementation) used to distribute monitoring data need not be the same that is used to distribute the application data. This prevents interference and allows for different deployment configurations.

7.3 Dependencies

7.3.1 HashId algorithm

This specification uses the `hashid` algorithm defined in DDS-XTYPES 1.3 (see [3]) to generate integer IDs from string names. To improve the specification readability the algorithm is copied below.
Given a string `<string_value>` the `hashid(<string_value>)` is computed as follows:

- Let **string_md5hash** be the MD5 Hash of the `<string_value>` encoded in `UTF-8`. This is a byte array.
- Let **hashid_tmp** be the `uint32` integer resulting from interpreting the first 4 bytes of **string_md5hash** as a Little Endian integer.
- Let **hashid** be equal to **hashid_tmp** `& 0x0FFFFFF`

For example, `hashid("data_reader") = 177233665`.

7.3.2 Supported Data Types

The Observable Data distributed by the Monitoring infrastructure must have an associated type to allow proper encoding and interpretation. The types supported by DDS Monitoring are a subset of the IDL types, specifically:

- Primitive Types
  - Numeric Primitive Types
  - Non-Numeric Primitive Types
- Primitive Collection Types
- Composite Types

These types are defined in the sections below.
Note that these are a subset of the types allowed in the IDL4, in particular unions, valuetypes, and interfaces are not supported.

7.3.2.1 Numeric Primitive Types

These are defined to be:

- The IDL primitive integer types :
  - `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, and `uint64`
- The IDL floating point types:
  - `float` and `double`.

7.3.2.2 Non-Numeric Primitive Types

These are defined to be:

- The IDL Discrete Types:
    - o `enum`, `bitmap`, and `bitset`
- The IDL primitive Non-Numeric Types:
    - o `octet`, `boolean`, `char`, and `wchar`.
- The IDL string types:
    - o `string` and `wstring`

7.3.2.3 Primitive Types

Types that are either Numeric Primitive Types (see 7.3.2.1) or Non-Numeric Primitive Types (see 7.3.2.2).

7.3.2.4 Collections of Primitive Types

These are defined to be:

- IDL `sequences` and `arrays` whose element type is a Primitive Types (see 7.3.2.3).
- IDL `maps` whose key type and value are Primitive Types (see 7.3.2.3).

7.3.2.5 Collections of Non-Primitive Types

IDL sequences, arrays, and maps of any supported data type that do not qualify as Collections of Primitive Types (see 7.3.2.4).

7.3.2.6 Collections Types

Types that are either Collections of Primitive Types (see 7.3.2.4) or Collections of Non-Primitive Types (see 7.3.2.5).

7.3.2.7 Structured Types

IDL `struct` types, whose members of any supported data type, including (recursively) Structured types.

7.4 Monitoring Resource Model

7.4.1 Overview

The Monitoring Resource Model provides a way to organize and contextualize the observable elements of a software stack:

- It provides a way to classify, identify, and name the application-relevant objects that are considered the source of the monitoring data.
- It organizes and names the observable elements of each resource and provides the association from those elements to the Monitoring Data produced.

The remaining sections define the building blocks of the Monitoring Resource Model.

7.4.2 Resources

A Resource is an abstraction representing the entities that can be monitored in the system. Resources generate Monitoring Data that can be observed and provide the context for the monitoring data to be interpreted.

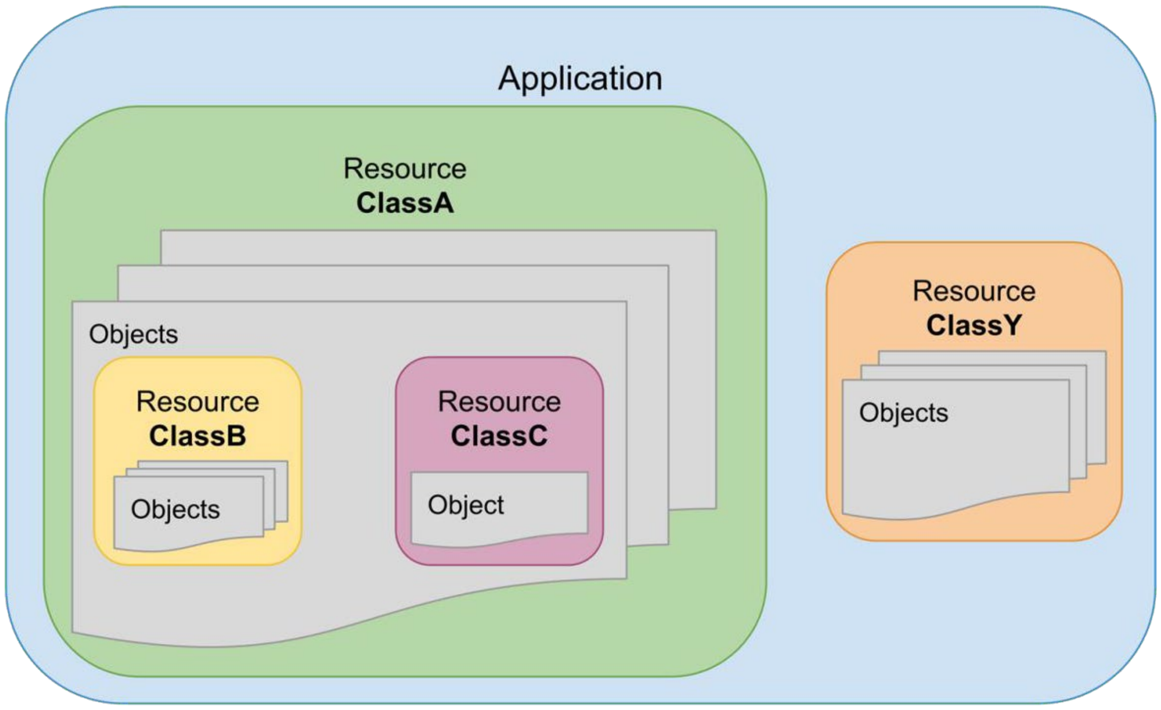


Figure 8.2: Application Information modeled as a set of related Resources

7.4.2.1 Resource Class

Resources have an associated Type, called the Resource Class. For any given Resource Class there can be multiple Resource Objects of that class.

For example, in the case of a DDS-based application, we may define a “data_writer” Resource Class, this resource type contains data elements whose value represents the status of that DDS DataWriter, such as the number of samples sent, the number of bytes sent, the number of repair messages sent, the number of matched DDS DataReaders, and so on.

A complete system may contain many Resource Objects whose Resource Class is “data_writer,” one for each DDS DataWriter Entity in the system.

7.4.2.1.1 ResourceClassName

Resource types are identified by a string called the *ResourceClassName* which may be different from the name of the type and is used to declare relationships between resources, see 7.4.2.2.

For example, in a DDS System the ResourceClassName “data_writer” may be used to identify the type of resource associated with a DDS DataWriter.
The ResourceClassName has global scope across all resource types, it is not scoped by the associated data type or any module the data type belongs to. It is explicitly assigned using the `@resource` annotation, see 7.4.9.1.7.

7.4.2.1.2 ResourceClassNamespace

Resource Classes must be assigned a namespace. The *ResourceClassNamespace* helps categorize the Monitoring Data generated by the resources of that class.
For example, all resources in DDS Monitoring are assigned the namespace "dds". This namespace is later used as part of *ObservableElementName* (see 7.4.3.4.1). That way the metrics produced by DDS resources are organized and differentiated from those coming from other facilities.
The *ResourceClassNamespace* has global scope across all resource types, it is not scoped by the associated data type or any module the data type belongs to. It is explicitly assigned using the `@resource` annotation, see 7.4.9.1.7.

7.4.2.1.3 ResourceClassId

Resource Classes are also identified by a *ResourceClassId* which is a 32-bit integer. The *ResourceClassId* is derived from the *ResourceClassName* using the `hashid` algorithm (see 7.3.1).

```
ResourceClassId = hashid(ResourceClassName)
```

For example, the resource class "domain_participant" has the `ResourceClassId`:

```
hashid("domain_participant")= 99258059
```

7.4.2.2 Resource Tree

Resources types can be related to other resource types by three kinds of relationships: "owner", "requires", and "uses".

- **Owner**. Each Resource Class may designate another resource Class as its "Owner". This is a containment-type relationship for the objects of the respective classes.
  - Objects of the "owned" resource exist within the context of their owner.
  - The owner relationship is also used to provide a scope for naming the resource Objects.
  - There can be at most one Owner for each Resource Class. A resource without an "Owner" is considered a "root" resource.
- **Requires**. Each Resource may designate one or more resources as "required". This means that Objects of that resource type need the existence of other objects of the specified classes.
- **Uses**. Each Resource may designate one or more resources as "uses". This means that Objects of that resource type may reference objects of the specified classes.

When a resource object is created, these class relationships result in corresponding relationships between the Resource Objects
For example, in a DDS System, a "topic" resource would designate the "domain_participant" resource as its "owner" as DDS Topics are created in the scope of a specific DomainParticipant. Likewise, the "topic" resource may designate the "requitered_type" resource as "required" given that in DDS a Topic must have an associated data type that has been registered with the DomainParticipant.
The *Resource Tree* is defined as the tree created by the "owner" relationship amongst the Resource Objects that exist in a given system.
Resources also contain Data elements (or members) whose value represents the status or configuration of the resource. See 7.4.3.
Root resources shall have a namespace explicitly assigned using the `@resource` annotation. Children resources can omit the explicit specification of their namespace. In this case, they are considered to belong to the same namespace as the parent resource.

7.4.2.3 Resource Object Identification

Resource Objects have a Global Unique Identifier (*ResourceGUID*) and a Global Resource Name (*ResourcePathName*) that allows them to be referenced in a human-readable way.

7.4.2.3.1 ResourceGUID

The *ResourceGUID* is assigned automatically by the Monitoring infrastructure. It is constructed using a compact binary representation. It provides an efficient way to identify and relate resources. The algorithm to generate it is vendor-specific, but it shall be constructed to be globally unique within the entire system being monitored.

7.4.2.3.2 ResourcePathName

The *ResourcePathName* is constructed from the *Resource Tree*. 7.4.2.2. It is a human-readable representation that uses string. The string uses a file "path" format that mirrors the hierarchical relations in the Resource Tree: Given the *ResourcePathName* the Resource can be easily located within the tree. Ideally, it should also be globally unique, but this cannot be enforced. The correct operation of the system does not depend on the *ResourcePathName* uniqueness since anything that requires unique identification shall use the *ResourceGUID*.
The algorithm to construct the *ResourcePathName* shall be as described below:

- Starting with the sting **resource_name** being the empty string, traverse the Resource Tree from the root until reaching the Resource Object being named.
- For each (resource) node traversed:
- Let `<resource_class_name>` be the *ResourceClassName* of the resource. This is assigned using the `@resource` annotation, see 7.4.9.1.7
- Let `<resource_object_name>` be a name assigned at the time the resource object is instantiated. This name must be unique within the scope of the Parent Resource Object.
- Append the 4 strings: "/", `<resource_class_name>`, `"s/"`, and `<resource_object_name>` to **resource_name**. Note the use of the character '/' as a separator and the extra "s" following the `<resource_class_name>`.

Note that `<resource_object_name>` is generally provided by the user who ultimately instantiates the resource object in their system. In the case of DDS, it would be the user that defines the DDS Entities and deploys them.
Generating a `<resource_object_name>` for the root resources that are unique within the scope of the Parent Object may present a challenge. This is because the scope for these root resource object names is the whole system being monitored.

- One approach could be to use some application-specific GUID that may be available when the application is deployed.
- In the case of DDS. The DomainParticipant GUID the DDS infrastructure generates for every DDS DomainParticipant.
- An alternative approach may combine host identifiers (e.g. hostname, MAC address, IP address) with processIds and type stamps.
- If a system is defined using DDS-XML, or a tool-based model-driven approach is followed, the entire distributed application can be expressed in a single model and unique names could be generated from it.
- If DDS-Security is being used, the Subject Names of the Identity certificates that uniquely identify each DomainParticipant may be used to create globally unique names.

7.4.2.3.3 Example

For example, a *ResourcePathName* used to identify a specific DDS DomainParticipant in the system may be:

```
/applications/ShapeApp_25/domain_participants/SquarePublisher
/applications/ShapeApp_25(GUID=86FGA845-8F24aD74)/domain_participants/SquarePublisher
/applications/ShapeApp_25(ip=142.250.189.196;ts=2024-05-17T20:26:26Z)/domain_participants
/SquarePublisher
```

7.4.3 Observable Elements

Resource Objects contain Data elements (or members) whose value represents the status or configuration of the resource.
The Data elements whose value can be observed are called *Observable Elements*. See 7.4.9 for the description of how the observability of data elements is specified.

7.4.3.1 Observable Element data type

Observable Elements have an associated data type. In DDS monitoring the type of *Observable Elements* must be one of the supported types defined in 7.3.2.

7.4.3.2 Observable Element Children

Observable Elements may contain children Observable Elements. This will be the case if the type associated with an *Observable Element* is a Structured Type (see 0). The Structured Type is considered the "parent" of its children's Observable Elements.
Observable Elements whose type is not a Structured Type (see 0) are considered "terminal" in the sense that they do not contain children Observable Elements.

- This applies trivially to Observable Elements whose type is a Primitive Type.
- This also applies to Observable Elements whose type is a Collection Type (see 7.3.2.6).

7.4.3.3 Resource Observable Element Tree

The *Resource Observable Element Tree* (shortened as the *Observable Element Tree*) is a tree of Observable Elements defined for each Resource Object that has the Resource Object itself as its root.
The Observable Element Tree contains all the Observable Elements that can be reached from the Resource Object, (recursively) following the parent-children relationship of the Structured data type associated with each Observable Element.
For example, assuming the Resource Class "application" is defined as:

```
module monitoring {
  module dds {
    @appendable @nested
    @observable_unit(distribution=PERIODIC)
    struct ProcessMemoryUtilization {
        @unit("B") uint64 resident_memory_bytes;
        @unit("B") uint64 virtual_memory_bytes;
    };

    @appendable @nested
    struct ProcessPlatformUtilization {
        @observable @unit("%"). uint16        cpu_usage;
        @observable ProcessMemoryUtilization  memory_usage;
    };
```

```
        @mutable  @nested
        @resource(class="application", namespace="dds")
        struct Application   {
            @observable(distribution=ON_CHANGE) string     hostname;
            @observable        ProcessPlatformUtilization  process_utilization;
        };
      };
    };
```
Then given a Resource Object for the "application" resource class, the corresponding Observable Element Tree would contain the data elements:
```
      .
      |- hostname
      |- process_utilization
           |- cpu_usage
           |- memory_usage
                |- resident_memory_bytes
                |- virtual_memory_bytes
```

7.4.3.4 Observable Element Identification

Observable Elements have an *ObservableElementName* and an *ObservableElementId*. Either one may be used to uniquely identify an *ObservableElement* within the scope of its owning Resource.

7.4.3.4.1 ObservableElementName

The *ObservableElementName* is a string that identifies each *Observable Element* within the scope of the Resource Object that contains it.
The *ObservableElementName* string is the concatenation of three strings: *ResourceClassNamespace* (see 7.4.2.1.2), *ResourceClassName* (see 7.4.2.1.1), and *ElementPathSuffix*. The resulting string is also converted to lowercase:
The *ElementPathSuffix* concatenates the member names of all the Observable Elements traversed following the *Observable Element Tree* starting at Resource Class until reaching the Observable Element.
All concatenations shall use the underscore character '_' as a separator. The '_' separator is not used if one of the strings being concatenated is the empty string.
Note that the `@obsevable_name` annotation may be used to substitute the member names with other strings (including the empty string) when constructing the *ElementPathSuffix* (see 7.4.9.1.6).
Note that the *ResourceClassNamspace* and *ResourceClassName* are not strictly needed to uniquely identify the *Observable Element* within the owning resource. However, the use of this prefix improves the usability of the Monitoring Administration API and the organization of metrics when mixed with those originating from other facilities.

7.4.3.4.2 ObservableElementId

The *ObservableElementId* is computed from the *ObservableElementName* using the `hashid` algorithm, see 7.3.1,
```
   ObservableElementId = hashid(ObservableElementName)
```

7.4.3.4.3 Example

Assuming the same resource model example in 7.4.3.3, the *ObservableElementName* of the observable elements would be:
```
   "dds_application_hostname"
   "dds application_process_utilization"
   "dds application_process_utilization_cpu_usage"
   "dds_application_process_utilization_memory_usage
   "dds_application_process_utilization_memory_usage_resident_memory_bytes"
```

```
"dds_application_process_utilization_memory_usage_virtual_memory_bytes"
```
In this example, the *ResourceClassNamespace* is `"dds"`. and the *ResourceClassName* is `"application"`.
The corresponding *ObservableElementId* of the observable elements would be:
```
166613811, 264598945, 205002779, 230327161, 137131145, 120161950.
```

7.4.4 Observable Unit

ObservableUnits are the subset of *ObservableElements* that can be sent atomically using DDS Monitoring. They represent the smallest unit of "distribution" or "network transmission".
ObservableUnits may be explicitly marked in the resource model, see 7.4.9, or may be deduced from the Observable Element Tree.
Being Observable Elements, the *Observable Units* also have an associated data type, see 7.3.2.
If the Type associated with an *Observable Unit* is a Structured Type (see 0), it may contain one or more (children) Observable Elements. In this case, the children observable elements cannot be sent separately from each other by DDS Monitoring. Whenever one needs to be sent all the other ones will also be sent.
The reason to define *Observable Units* with structure types is to improve performance and minimize resource utilization. These can be especially important when the monitored systems are real-time and/or edge systems.
If the Type associated with an *Observable Unit* is a Collection Type, the elements of the collection may have Structured data types. Despite this, the *Observable Unit* is treated as a unit of selection and "distribution".

7.4.4.1 Observable Unit Identification

Observable Units are *ObservableElements* so they are identified by their *ObservableElementName* and an *ObservableElementId*. Either one may be used to uniquely identify an *ObservableUnit* within the scope of their owning Resource Object.

7.4.5 Metrics (Numeric Primitive Types)

Metrics represent the *ObservableElements* containing Numeric Primitive Types (7.3.2.1).
Metrics are used to hold a 'measurement' of some aspect of the status of the resource. They contain the kind of information that may be stored as a time series so that statistics and trends may be analyzed. They are the natural interface to Telemetry backends focused on storing and analyzing time-series data.
For example, a resource representing an Operating System Process may have metrics about hardware utilization, such as, "current CPU usage" and the "current memory usage."

7.4.5.1 Metrics vs Observable Units

The term *Metric* is a convenient and intuitive way to refer to **all** the *ObservableElements* in the Resource Tree that have a Numeric Primitive Types.
Some *Metrics* are also *ObservableUnits*. Other *Metrics* are grouped with other *Metrics* all nested inside an *ObservableUnit*:

- *ObservableUnits* whose associated data type is a Numeric Primitive Type are also Metrics.
- *ObservableUnits* whose associated data type is a Structured Type (see 7.3.2.7) may contain multiple Metrics inside, one per nested element whose type is a Numeric Primitive Type.

In cases where a *Metric* is not an *ObservableUnit*, rather it is only a part of it, alongside other *Metrics*, it is not possible to receive that *Metric* by itself, It will be received alongside all the other metrics in the Observable Unit. However, the application receiving the Observable Unit may request and address the individual *Metrics* when configuring and processing the Monitoring Data.
For example, assume the simplified resource definition below:

```
@appendable @nested
@observable_unit(distribution=PERIODIC)
struct ProcessMemoryUtilization {
    @unit("B") uint64 resident_memory_bytes;
    @unit("B") uint64 virtual_memory_bytes;
};

@appendable @nested
struct ProcessPlatformUtilization {
    @unit("%"). @observable uint16        cpu_usage;
    @observable ProcessMemoryUtilization  memory_usage;
};
@mutable @nested
@resource(class="application")
struct Application   {
    @observable(distribution=ON_CHANGE) string     hostname;
    @observable        ProcessPlatformUtilization  process_utilization;
};
```

In this example, the observable element named "`dds_application_process_utilization_cpu_usage`" would be an *ObservableUnit*. The reason is that its type is a Primitive Type and none of the containing Observable elements has the `@observable_unit` annotation (the algorithm to define *ObservableUnits* from the resource model is described in 7.6). Since the data type is a Numeric Type (`uint16`) it is also a *Metric*.
Continuing the example, the observable element named "`dds_application_process_utilization_memory_usage`" would also be an *ObservableUnit* (see 7.6) because the associated type (`struct ProcessMemoryUtilization`) has the `@observable_unit` annotation. Note that since this last element is not a Numeric Primitive Type it is not a *Metric*. Rather, it contains two metrics corresponding to its two Numeric Primitive Type members: `resident_memory_bytes` and `virtual_memory_bytes`.

7.4.5.2 Metric Identification (MetricId, MetricName)

Metrics are *ObservableElements* so they are identified by their *ObservableElementName* and an *ObservableElementId*. Either one may be used to uniquely identify a *Metric* within the scope of their owning Resource.

7.4.6 Attributes

Attributes represent the *ObservableElements* containing Non-Numeric Primitive Types (see 7.3.2.2) or Collection Types (see 7.3.2.6).
Attributes are used to represent configuration, classification., or contextual data, encoding additional aspects of the status of a resource.
For example, a Resource representing an Operating System Process may contain multiple attributes, such as the `executable_name` and the `user_name` of the parent process.

7.4.6.1 Attributes vs Observable Units

The term *Attribute* is a convenient and intuitive way to refer to **all** the *ObservableElements* that either have Non-Numeric Primitive Types in the Resource Tree or have a numeric type but should be treated as Non-numeric as indicated by annotations in the Monitoring Resource Model.
Some *Attributes* are also *ObservableUnits*. Other *Attributes* are grouped with other *Attributes* all nested inside an *ObservableUnit*:

- *ObservableUnits* whose associated data type is a Non Numeric Primitive Type (see 7.3.2.2 are also *Attributes*.
- *ObservableUnits* whose associated data type is a Numeric Primitive Type (see 7.3.2.1) and have the annotation `@attribute` (see 7.4.9.1.2) are also *Attributes*.
- *ObservableUnits* whose associated data type is a Collection Type (see 7.3.2.6) are also *Attributes*.
- *ObservableUnits* whose associated data type is a Structured Type (see 7.3.2.7) may contain multiple *Attributes* inside.

Similar to *Metrics*, in cases where an *Attribute* is not an *ObservableUnit*, rather it is only a part of it, alongside other *Attributes*, it is not possible to receive that *Attribute* in isolation, without also receiving all the other *Attributes* in the Observable Unit.
For example, the observable unit named "`dds_application_hostname`" has an associated data type `string`. So it is also an *Attribute*.
For example, the observable unit named "`dds_publisher_qos_presentation`" has an associated structure data type `PresentationQosPolicy`, shown below, so it is not an *Attribute*.

```
@appendable @nested
@observable_unit(distribution=ON_CHANGE)
struct PresentationQosPolicy {
    PresentationQosPolicyAccessScopeKind access_scope;
    boolean coherent_access;
    boolean ordered_access;
};
```

Based on the above definition, the observable unit "`publisher_qos_presentation`" contains three Attributes corresponding to each member of the `PresentationQosPolicy`.

7.4.6.2 Attribute Identification

Attributes are *ObservableElements* so they are identified by their *ObservableElementName* and an *ObservableElementId*. Either one may be used to uniquely identify the *Attribute* within the scope of their owning Resource.

7.4.7 Logs and Events

Resources may also generate observable data in the form of Logs and Events. This type of observable data is a record describing a discrete occurrence within a Resource.

- **Events** represent significant occurrences within systems that are specifically captured because of their relevance to the system's operation or behavior. Data representing Events is typically structured with specific fields to convey information about the incident.

- **Logs** represent a broader category that can include anything from detailed debug information to high-level system alerts and errors. They are useful for troubleshooting, providing a detailed trail of what happened where, and when. Data representing logs typically has standard fields used for classification as well as formatted strings containing more detailed information.

This specification does not distinguish Events from Logs, treating events as another kind of log. Furthermore, it uses the same model for Logs as DDS-Security which itself follows the Syslog model (IETF RFC 5424).

7.4.7.1 Log Identification

Logs are identified within the context of their owning Resource by the combination of the *SyslogFacility* and a *LogSequenceNumber*.

The *SyslogFacility* is a standard numerical code that represents the source of the log message, allowing the observer to categorize and filter log messages based on their origin. These codes are defined in IETF RFC 5424, see [7].

This specification uses:

- Code 23 (MIDDLEWARE) for logs generated by Resources representing DDS Entities.
- Code 22 (SERVICE) for logs generated by Resources representing DDS Services (e.g. Persistence Service).
- Code 10 (SECURITY_EVENT) for any logs generated by DDS Security Plugins.
- Code 1 (USER) for logs generated by other Resources.

7.4.8 Monitoring Data

Monitoring Data is any data distributed by the Monitoring infrastructure. It includes data generated by sampling Observable Elements as well as Logs and Events.
Resources are the source of all the Monitoring Data distributed by the Monitoring Infrastructure.

7.4.9 IDL definition of the Monitoring Resource Model

Within the context of DDS Monitoring, the Monitoring Resource Model consists of the definition of the Resource Classes, including relationships between resources (owns, requires, uses). It also includes the definition of the Observable Elements and Observable Units. Beyond that, the model also includes additional information that may impact the naming or distribution of Observable Elements.
This DDS-Monitoring specification uses OMG IDL to formally define the DDS Monitoring Resource model. The DDS-Monitoring specification defines custom IDL annotations that identify resource classes, and all the additional information required to fully specify the resource model. IDL annotations are also used to configure the distribution aspects of the monitoring data.
These annotations could be used to define Monitoring Resource models for other kinds of software systems, not just DDS-based systems.

7.4.9.1 Annotations used to define the Monitoring Resource Model

The custom annotations defined by the DDS-Monitoring speciation are defined in the IDL file `monitoring_annotations.idl` included as part of this specification. This clause specifies how they are used.

7.4.9.1.1 @view

This annotation may be applied to members of a Structured Type. The annotation controls the serialization of data objects of that type, allowing the definition of multiple levels of "detail" in the serialization.
The annotation is defined in the following IDL (all definitions are in the module monitoring::dds):

```
@annotation view {
    @max(31) uint8  level          default 0;  // 0 => default level
    @max(31) uint8  member_level  default 0;  // member default level
    };
```

The `@view` annotation on a member of a Structure type may be used to define multiple ways to serialize the structure. Each of these is considered a "level" in the sense that it serializes additional members beyond the ones serialized by the level below.
The `@view` annotation on a member of a Structure type may also be used to select a specific serialization level for serializing the member.
Each different value of the `level` parameter defines a "serialization level" for the Structure type. The serialization level is identified by the value of the `level`.
Non-structure types are considered to have only one level, selected by level 0.
The serialization of a data object of type a Structure Type for level **LI** is done according to the rules below:

- Members without the `@view` annotation are treated as if they had the annotation `@view(level=0, member_level=0)`.
- Members having the `@view` annotation with the parameter `level` <= **LI** are serialized. Other members are treated as if they had the `@non_serialized` annotation.
- The serialization of a member that has the `@view` annotation serializes the member according to the member type for the level identified by the `member_level`.

7.4.9.1.1.1 Parameters

The annotation may specify the following parameters:

- Parameter **level**. Selects the members included in the level identified by the **level** value. The level identified by **level**=**LI** contains only the members with the `@view` annotation that have parameter **level** <= **LI**.

- Parameter **member_level**. Specifies the level used when serializing the member.

7.4.9.1.1.2 Example

The following IDL example illustrates the use of the `@view` annotation:

```
@appendable @nested
struct FloatStat {
    @view(level=2) uint32 period_ms;
    @view(level=2) uint64 count;
    @view(level=0) float mean;
    @view(level=1) float min;
    @view(level=1) float max;
};

@nested
@mutable
struct ParticipantPeriodic {
    @view(level=0, member_level=1) @optional
    FloatStat send_samples_per_s;
    @view(level=1, member_level=0) @optional
    FloatStat send_bytes_per_s;
    @view(level=2, member_level=2) @optional
    FloatStat receive_samples_per_s;
    @view(level=2, member_level=2) @optional
    FloatStat receive_bytes_per_s;
}
```

The type `FloatStat` above defines 3 serialization levels identified by `level` values 0, 1, and 2.

- Level with level = 0: serializes member `mean`.
- Level with level = 1: serializes the same members as level 0 plus `min` and `max`.
- Level with level = 2: serializes the same members as level 1 plus `period_ms` and `count`.

The type `ParticipantPeriodic` defines 3 serialization levels identified by level values 0, 1, and 2.

- Level with level = 0: serializes member `send_samples_per_s`.
- Level with level = 1: serializes member `send_samples_per_s`. and `send_bytes_per_s`
- Level with level = 2: serializes member `send_samples_per_s`, `send_bytes_per_s`, `receive_samples_per_s`, and `receive_bytes_per_s`.

These members are serialized as follows:

- Member `send_samples_per_s` serializes level **1** of the `FloatStat` type. Therefore, it serializes `mean`, `min`, and `max`.
- Member `send_bytes_per_s` serializes level **0** of the `FloatStat` type. Therefore, it serializes `mean`.
- Member `receive_samples_per_s` serializes level **2** of the `FloatStat` type. Therefore, it serializes `period_ms`, `count`, `mean`, `min`, and `max`.
- Member `send_bytes_per_s` serializes level **2** of the `FloatStat` type. Therefore, it serializes `period_ms`, `count`, `mean`, `min`, and `max`.

Just because a Member type defines multiple serialization levels it does not mean the container type has to also define multiple levels. The example `ParticipantPeriodic` defined earlier in this clause only has one level (level 0) despite containing members whose type defines multiple levels.

7.4.9.1.2 @attribute

This annotation may be applied to members of a Structure Type. The annotation indicates that any Observable Elements related to the member (directly or nested within) that are Primitive Types should be treated as Attributes. The use of the annotation overrides the default criteria used to make the decision.
The annotation is defined in the following IDL (all definitions are in the module monitoring::dds):

```
@annotation attribute {
};
```

By default the decision of whether an Observable Element becomes a Metric or an Attribute is made based on the type associated with the element:

- Observable Elements whose type is a Numeric Primitive Type (7.3.2.1) become Metrics.
- Observable members whose type is a Non-Numeric Primitive Type (7.3.2.2), become Attributes.

If this annotation is used, the default criteria is modified and any Observable Elements related to the marked member (the member itself and any nested members) will become Attributes even if their type is a Numeric Primitive Type.

7.4.9.1.2.1 Parameters

The `@atribute` annotation does not have any parameters.

7.4.9.1.2.2 Example

The following IDL annotations provide a simple definition of a Resource Class "simplified_participant".

```
@final @nested
struct RTPSVersion_t
    uint8 major;
    uint8 minor;
};

@mutable @nested
@resource(class="simplified_participant", owner="application")
struct SimplifiedParticipant   {
        @observable  uint64        messages_sent;
        @observable  string        domain_tag;
        @attribute
        @observable  uint32        domain_id;
        @attribute
        @observable  RTPSVersion_t  rtps_version;
    };
```

The observable element `messages_sent` will be classified as a *Metric* according to the default criteria given its type is a Numeric Primitive Type (7.3.2.1).

Likewise, the observable element `domain_tag` will be classified as an *Attribute* according to the default criteria as its type is a Non-Numeric Primitive Type (**Error! Reference source not found.**).

The member `domain_id` would have been classified as a *Metric* according to the default criteria as its type is a Numeric Primitive Type. However, the presence of the `@attribute` annotation changes the behavior, and it is classified as an *Attribute* instead.

The observable element `rtps_version` results in two Attributes: `rtps_version_major` and `rtps_version_minor`. Although the nested observable elements have integer types, the presence of the `@attribute` annotation changes the behavior and they become *Attributes* instead.

7.4.9.1.3 @observable

This annotation may be applied to a member of a Structure type. The annotation controls whether the member becomes an Observable Element when it appears in a Resource Observable Element Tree.

The annotation is also used to configure distribution aspects of nested Observable Elements, see 7.5, specifically whether they are sent periodically (see 7.5.1) or only when there are (significant) changes (see 7.5.2).

The annotation is defined in the following IDL (all definitions are in the module monitoring::dds):

```
    enum DistributionKind {
        UNSPECIFIED,
        PERIODIC,
        ON_CHANGE
    };

    @annotation observable {
        DistributionKind    distribution    default UNSPECIFIED;
    };
```

For Structures representing the resources, there shall always be at least one member with the `@observable` annotation.

The annotation may be applied to only a subset of the members of a Structure Type. In this case, any member that does not have the annotation **will not** be considered an Observable Element, unless the annotation `@observable_view` has also been applied to the enclosing Structure Type. See 7.4.9.1.5.

The special case where no member is annotated is treated as if all members had the annotation `@observable(distribution=UNSPECIFIED)` for structure types that are not a Resource Class.

7.4.9.1.3.1 Parameters

The observable annotation may specify the following parameters:

- Parameter **distribution**. Configures aspects of the distribution of the Observable Elements. It may take three values:
    - PERIODIC. This setting modifies the default distribution of data for nested Observable Elements to PERIODIC, see 7.5.1.

- ON_CHANGE. This setting modifies the default distribution of data for nested Observable Elements to event-driven (ON_CHANGE), see 7.5.2.
- UNSPECIFIED. This setting does not modify the default distribution of data for nested Observable Elements.

7.4.9.1.3.2 Algorithm to determine the DistributionKind of Observable Elements

Observable Elements may appear nested inside *Observable Elements* and those could have `@observability` annotations. In addition, for *Observable Elements* whose type is a Structured type, the containing Type itself could have `@observability` annotations.
The Monitoring Distribution Model (7.6) needs to determine the `DistributionKind` of *Observable Element* whose associated type is a Primitive Type (7.3.2.3) or a Collection Type (7.3.2.6). This determination is based on the Resource's *Observable Element Tree*. The rules shall be applied in the order they appear:

- Do a depth-first Resource's *Observable Element Tree*, setting the initial *parentDistribution* to UNSPECIFIED.
- When visiting an *Observable Element* node in the tree*:*
- If the *Observable Element* type is an Observable Unit (see 7.4.4), the distribution kind shall be set according to these rules:
  - If the Observable Element has an `@observable` annotation with a *distribution* parameter value different from UNSPECIFIED, use the value of *distribution* in the annotation.
  - Otherwise, if the value of the *parentDistribution* is different from UNSPECIFIED, use the value of the *parentDistribution*.
  - Otherwise, if the Observable Element type is a Structured Type and the `@observable_unit` annotation *distribution* parameter has a value different from UNSPECIFIED, use the value of the *distribution* parameter in the Structure Type.
- Otherwise, set the `DistributionKind` to ON_CHANGE.
- Otherwise, the *Observable Element* type must be Structured Type:
- If the *Observable Element* has `@observable` annotation and the value of the *distribution* parameter is not UNSPECIFIED, set the value of the *parentDistribution* to match the *distribution* in the annotation.
- Recurse through into the Structure Type, visiting each nested Observable Element.
- Once the recursion completes, set the *parentDistribution* back to the value it had before visiting the *Observable Element*.

7.4.9.1.3.3 Example

Assume the following definition of a simplified application resource:

```
 @observable_unit(distribution=PERIODIC)
 struct MemoryStatus   {
     uint64  bytes_used;
     uint64  allocation_cummulative_count;
     uint64  free_cummulative_count;
};

 struct NetworkStatus   {
     @observable distribution=ON_CHANGE)   string         nic_name;
     @observable(distribution=ON_CHANGE)   uint64         nic_speed;
     @observable                           uint64         messages_sent;
};

 struct ProcessState   {
     @observable                           string         cpu_model_name;
     @observable(distribution=ON_CHANGE)   uint32         pid;
     @observable                           MemoryStatus   memory;
     @observable(distribution=PERIODIC)    NetworkStatus  network;
     @observable(distribution=PERIODIC)    uint64         cpu_time;
```

```
    @observable                         uint8          percent_cpu_use;
    string                                             command;
};

@mutable @nested
@resource(class="simplified_application")
struct SimplifiedApplication   {
    @observable(distribution=ON_CHANGE) string        hostname;
    @observable                         ProcessState  process_state;
};
```

The algorithm traverses the Observable Element tree:

Set *parentDistribution* to UNSPECIFIED before starting the traversal.

The observable element `hostname` is a Primitive Type and specifies *distribution=ON_CHANGE* using the `@observability` annotation directly in the element, so the `DistributionKind` is set to match the annotation: ON_CHANGE.

The observable element `process_state` is a Structure. It has a `@observability` but it does not have a *distribution* parameter so it is interpreted as being UNSPECIFIED, which means that *parentDistribution* is not modified: it remains set to UNSPECIFIED.

Then the traversal visits the Observable Elements contained by ***process_state*** these are obtained from the associated member type: `ProcessState`:

- Element `cpu_model_name` is a Primitive Type. It does not have an `@observability` y annotation so it checks the *parentDistribution*. Since this is UNSPECIFIED it checks if the type `ProcessState` has the `@observable` or `@observable_unit` annotations, since it does not have them it sets the `DistributionKind` to the default (ON_CHANGE).
- Element `pid` is a Primitive Type, it has an `@observability` annotation with a specified *distribution* so it sets the distribution kind accordingly. It is set to ON_CHANGE.'
- Element `memory` is a Structure type (`MemoryStatus`). It has a `@observability` but it does not specify the *distribution*, therefore the *parentDistribution* is not modified: it remains set to UNSPECIFIED. The traversal visits the elements inside `MemoryStatus`.
- Element `bytes_used` is a Primitive Type. It does not have an `@observability` annotation so it checks the *parentDistribution*. Since this is UNSPECIFIED it checks if the type `MemoryStatus`. This type has the `@observability_unit` annotation with parameter *distribution*=PERIODIC so the `DistributionKind` of `bytes_used` is also set PERIODIC.
- Likewise, the `DistributionKind` of `allocation_cummulative_count` and `free_cummulative_count` are also set to PERIODIC.
- Element `network` is a Structured type (`NetworkStatus`). It has a `@observability` with parameter *distribution* = PERIODIC to the *parentDistribution* is set to PERIODIC. The traversal visits the elements inside `NetworkStatus`.
- Element `nic_name` is a Primitive Type. Its `DistributionKind` will be set to ON_CHANGE based on the `@observability` annotation having parameter *distribution*=ON_CHANGE.
- Element `nic_speed` is a Primitive Type. Its `DistributionKind` will be set to ON_CHANGE based on the `@observability` annotation having parameter *distribution*=ON_CHANGE.
- Element `messages_sent` is a Primitive Type. Its `DistributionKind` will be set to PERIODIC based on the *parentDistribution* as it does not have an `@observability` annotation.
- Exiting the `network` Observable Element sets the *parentDistribution* back to UNSPECIFIED.
- Element `cpu_time` is a Primitive Type. Its `DistributionKind` will be set to PERIODIC based on the `@observability` annotation having parameter *distribution*=PERIODIC.

- Element `cpu_time` is a Primitive Type. Its `DistributionKind` will be set to ON_CHANGE because it does not have `@observability` annotation and the *parentDistribution* is UNSPECIFIED, so it uses the default value of ON_CHANGE.
- Element `command` is not an Observable Element since it does not have the `@observability` annotation and is in a Structure type (`ProcessState`) with other elements that do have the `@observability` annotation.

7.4.9.1.4 @observable_unit

This annotation may only be applied to a Structured type. The annotation is used to define an Observable Unit (see 7.4.4). *Observable Elements* that appear (recursively) as children of the Observable Unit will not be propagated individually. Instead, they will appear grouped with the other Observable Elements in the unit.

The annotation may also impact the `DistributionKind` of nested Observable Elements, see 7.4.9.1.3.2.

This use of this annotation and the related `@observable_view` (see 7.4.9.1.5) impact the mapping of structure into the Monitoring distribution data model.

This grouping of *Observable Elements* simplifies the mapping to the distribution data model and can also increase performance in situations where multiple observable elements are often sent together. Both are important concerns for real-time or Edge systems.

The annotation is defined in the following IDL (all definitions are in the module monitoring::dds):

```
@annotation observable_unit {
    DistributionKind    distribution    default UNSPECIFIED;
};
```

The enumerated type `DistributionKind` is defined in 7.4.9.1.3.

If a Structured type does not have the `@observable_unit` annotation, then the nested *Observable Elements* shall be mapped into the Distribution Model in such a way that it is possible to send them separately.

If a Structured type has the `@observable_unit` annotation, then the *Observable Elements* nested within shall be mapped into the Distribution Model grouped together. This means that if one of the *Observable Elements* needs to be sent, other *Observable Elements* placed in the same unit will also be sent, even if not explicitly requested by the consumer.

The number of *Observable Units* used to send the nested *Observable Elements* depends on whether the annotation `@observable_view` is also used.

7.4.9.1.4.1 Parameters

The observable annotation may specify the following parameters:

- Parameter **distribution**. Configures aspects of the distribution of the monitoring data. It may take the same three values described in 7.4.9.1.3.1. The setting impacts all the contained observable elements in the unit.

7.4.9.1.4.2 Example

The following IDL annotations define some of the *Observable Elements* in a type used to represent the state of a Process.

```
@appendable @nested

@observable_unit(distribution=PERIODIC)

struct ProcessMemoryUtilization {

    @unit("B") uint64 resident_memory_bytes;

    @unit("B") uint64 virtual_memory_bytes;

};
```

The presence of the `@observable_unit` causes the two *Observable Elements* `resident_memory_bytes` and `virtual_memory_bytes` to be grouped together in the distribution data model such that one cannot be sent without also sending the other.

7.4.9.1.5 @observable_view

This annotation may only be applied to a Structured type that has the `@observable_unit` annotation. It may appear multiple times in the same Structure. Each occurrence in a type shall use a different value for the "level" parameter (see below).

The annotation is defined in the following IDL (all definitions are in the module monitoring::dds):

```
@annotation observable_view {
    @max(8) int8        level           default 0;
    string              select          default "%";
};
```

The annotation determines the number of *Observable Units* created from the Structure type and the *Observable Elements* included in each unit.

- If the annotation is not present the distribution model shall have a single unit containing all the nested *Observable Elements*.
- If the annotation is present the distribution model shall have one unit for each `@observable_view` annotation.
    - Given the annotation `@observable_view(level=`**`LI`**`, select=<pattern>)`, the corresponding unit shall contain all the nested *Observable Elements* that match the **select** <pattern> parameter in the annotation, in addition to all the nested *Observable Elements* contained in all the units for levels with level 0<= **level** < LI.
    - Each incremental value of the level adds the *Observable Elements* that match the corresponding select parameter.

7.4.9.1.5.1 Parameters

The annotation may specify the following parameters:

- Parameter **level**. Used to identify each of the observable units created from the Composite Type. The value is interpreted as an inclusion "level":
    - Increasing values of the **level** shall correspond to increasingly more observable element detail. The most common observable elements should be placed at level=0, the next set at level=1, and so on.
    - The distribution model groups the observable elements in a level with the observable elements in all lower levels. Therefore, to send a metric at level=L the distribution model will also send the metrics at level =L-1, and recursively to the metrics at level=0.
- Parameter **select**. Expression used to identify the (subset) of members of the Structure Type that will be added to the Observable Unit identified by the **level**.
    - The selection of the members of a level shall be done in the order that corresponds to increasing values of the **level** parameter.
    - The expression can contain one or more patterns separated by a semicolon ';' character.
        - Only members who are not already part of a unit with a lower value for the **level** parameter are considered for matching.
        - Each pattern is applied in order against the name of each member. If any pattern matches, the member will be considered part of the unit associated with the value of the **level**.

- Each pattern can contain only alphanumeric characters, and the characters '_' and '%'.
    - The character '%' is treated as a special character that can match any number of characters in the member's name.
    - Other characters in the pattern must match exactly with the corresponding characters in the member name.
- Examples:
    - The select expression "%" will match all members of an IDL structure.
    - The select expression "%_count" will match any members of an IDL structure with a name ending in the suffix "_count".
    - The select expression "min;max" will match any members of an IDL structure that have exactly the name "min" or "max".
    - The select expression "min_%;max_%" will match any members of an IDL structure that have a name that starts with the prefix "min_" or "max_".

7.4.9.1.5.2 Example

The following IDL annotations define three units with levels 0, 1, and 2 from the structured type `Int32Stat`.

```
@appendable @nested
@observable_unit
@observable_view(level=0, select="mean")
@observable_view(level=1, select="min;max")
@observable_view(level=2, select="%")
@appendable @nested
struct FloatStat {
    @view(level=2) uint32 period_ms;
    @view(level=2) uint64 count;
    @view(level=0) float  mean;
    @view(level=1) float  min;
    @view(level=1) float  max;
};
```

In this example:

- The definition of units starts with the lowest level (level =0). The corresponding select expression "mean" matches the one member that has that exact name. Therefore, the unit for level=1 contains the member:
    - mean (included because it matches the expression for level 0)
- The definition of the unit for level=1 matches the select expression "min;max" against all members (except for the member ***mean*** as it already belongs to level=0). The expression "min;max" ends up matching the two members with those exact names. Therefore, the unit for level=1 contains the following members:
    - mean (included in level 1 because it is part of level 0)
    - min (included because it matches the expression for level 1)
    - max (included because it matches the expression for level 1)
- The definition of the unit for level=2 matches the select expression "%" against all members that are not part of level 0 or 1. This ends up matching all remaining members (***period_ms*** and ***count***). Therefore, the unit for level=1 contains the following members:
    - period_ms (included because it matches the expression for level 2)
    - count (included because it matches the expression for level 2)
    - mean (included in level 2 because it is part of level 0)

    - min (included in level 2 because it is part of level 1)
    - max (included in level 2 because it is part of level 1)

7.4.9.1.5.3 Relationship with @observability and @view

The `@observable_view` can be considered a convenient "shorthand" notation for common uses of the `@observable` and `@view`.

The `@observable_view` annotation applied to a Structure type is equivalent to applying the `@observable` and `@view` annotations to some of the members of the structure:

For each `@observable_view(level=<level>, select=<pattern>)` that appears in the Structure is equivalent to adding the annotations `@observable` and `@view(level=0, member_level=<level>)` to all the members that have a member name that matches the `<pattern>` according to the rules described in 7.4.9.1.5.

For example, assume `FloatStat` structure type is defined as shown below using the `@observable_view` annotations on the structure:

```
@observable_view(level=0, select="mean")
@observable_view(level=1, select="min;max")
@observable_view(level=2, select="%")
struct FloatStat {
    uint32 period_ms;
    uint64 count;
    float  mean;
    float   min;
    float  max;
};
```

The above is completely equivalent to the `FloatStat` type below that uses `@observable` and `@view` annotations on the structure members:

```
struct FloatStat {
    @observable @view(level=2) uint32 period_ms;
    @observable @view(level=2) uint64 count;
    @observable @view(level=0) float  mean;
    @observable @view(level=1) float  min;
    @observable @view(level=1) float  max;
};
```

7.4.9.1.6 @observable_name

This annotation may be applied to a member of an IDL Structure. The annotation impacts the *ObservableElementName* (see 7.4.3.4) of the Observable Elements that correspond to the annotated member and its children.

The annotation is defined in the following IDL (all definitions are in the module monitoring::dds):

```
@annotation observable_name {
    string   value    default "";
};
```

As described in 7.4.3.4, the *ObservableElementName* is constructed from the Resource Observable Element Tree concatenating the *member names* on the Types that correspond to the Observable Elements. The presence of the `@observable_name()` annotation on a member replaces the *member name* used with the value specified in the annotation.

7.4.9.1.6.1 Parameters

The annotation may specify the following parameters:

- Parameter **value**. Used to modify the name used to construct FQNs, *MetricNames*, and *AttributeNames*.

- If **value** is not present or set to the empty string (`""`) the name of the Module or Member is omitted in the construction of FQN, MetricNames, and AttributeNames.
- If **value** is not set to a non-empty the specified string is used instead of the name of the module of member in the construction of FQNs, MetricNames, and AttributeNames.

7.4.9.1.6.2 Example

Assume the following resource definition:

```
module monitoring { module dds {

    @appendable
    struct ProcessPlatformUtilization {
        /* Elements removed for illustrative purpose */
        uint32                   uptime_sec;
    };

    @mutable @nested
    @resource(class="application", namespace="dds", owner="")
    struct Application   {
        /* Elements removed for illustrative purpose */
        @observable(distribution=PERIODIC) @observable_name("process")
        ProcessPlatformUtilization process_utilization;
        /* Elements removed for illustrative purpose */
    };};};
```

Without the `@observable_name` annotation applying the rules in 7.4.9.1.3.2, the *ObservableElementName* for the application resource element containing the process uptime would be `"dds_application_process_utilization_uptime"`.
However, the `application` resource member `process_utilization` has the annotation `@observable_name("process")`, therefore the *ObservableElementName* is `"dds_application_process_uptime"`.

7.4.9.1.7 @resource

This annotation may be applied to any structured type. It is used to define a Resource Class.

The annotation is defined in the following IDL (all definitions are in the module monitoring::dds:

```
module DDS { module Monitoring {
    @annotation resource {
        string   name;
        string   namespace      default "";
        string   owner          default "";
        string   requires       default "";
        string   uses           default "";
    };
};};
```

7.4.9.1.7.1 Parameters

The annotation may specify the following parameters:

- Parameter **class**. The name used to identify this resource type.
    - If `name` is not present or set to the empty string (`""`) the name of the structured type is used.
    - The name used for the class should be unique among all the resources that have the same **owner**.
- Parameter **namespace**. A namespace that helps organize and categorize the Monitoring Data generated by the resource Observable Elements. The namespace is used as part of the ObservableElementName.

    - The **namespace** shall be present if the resource is a root resource (i.e. it does not have an owner).
    - if the resource is not a root resource and the **namespace** is not present (or it is set to the empty string (`""`), then the namespace of the owner resource is used.
- Parameter **owner**. The name of the Resource Class that is the direct parent of the resource within the resource tree.
    - If **owner** is not present or it is set to the empty string (`""`) the Resource Class is directly under the root of the resource class tree. These are called "root resource classes". The name of each root resource class should be unique among all root resource classes.
- Parameter **requires**. The name of one or more Resource Classes that the resource depends on. If multiple resources are listed, each is separated from the previous by a semicolon (';') character.
    - Every Resource Object of the Resource Class shall be associated with a Resource Object of each of the classes listed in the **requires** parameter.
- Parameter **uses**. The name of one or more Resource Classes that the resource may use. If multiple resources are listed, each is separated from the previous by a semicolon (';') character.
    - Resource Object of the Resource Class may only be associated with Resource Objects of the classes that appear listed in the **uses** parameter, in addition to the ones that appear listed in the **requires** parameter.

7.4.9.1.7.2 Example

The following IDL annotations define a resource class called "topic" that contains the monitoring data associated with a DDS Topic Entity. All IDL definitions are in the module monitoring::dds:

```
@mutable @nested
@resource(class="topic", owner="domain_participant", requires="type")
struct Topic {
    @observable(distribution=ON_CHANGE) GUID_t          dds_guid;
    @observable(distribution=ON_CHANGE) ObjectName      topic_name;
    @observable(distribution=ON_CHANGE) string          registered_type_name;
    @observable                         TopicQos        qos;
    @metric_name("")
    @observable                         TopicStatus     status;
};
```

The "topic" resource objects will be nested inside resource objects belonging to the class "domain_participant" and require resource objects of class "type".

7.4.9.2 Full IDL definition of the Monitoring Resource Model

The full definition of the DDS Monitoring Resource model is provided in the following IDL files which are included as part of this specification:

- `monitoring_annotations.idl`
- `monitoring_statistics.idl`
- `monitoring_logging.idl`
- `monitoring_resource.idl`
- `monitoring_administration.idl`
- `monitoring_dds_common.idl`
- `monitoring_dds_status.idl`
- `monitoring_dds_qos.idl`
- `monitoring_dds_entities.idl`

The types in the above "monitoring_dds_*" IDL files are contained inside the module `monitoring::dds`. The types in the remaining IDL files are inside the module `monitoring::dds`. These types are independent of DDS and could be used as Resource models for non-DDS systems.

7.4.9.3 Description of main types in the Monitoring Resource Model

7.4.9.3.1 Type **`ResourceClassId`**

This type is used to identify the `Resource` type, see 7.4.2.1. It is defined as shown in the IDL below:

```
typedef uint32 ResourceClassId;
```

7.4.9.3.2 Type **`ResourceClassName`**

This type is used to identify the `Resource` type using a human-readable representation, see 7.4.2.1. It is defined as shown in the IDL below:

```
const int32 RESOURCE_CLASSNAME_LENGTH_MAX        = 63;
typedef string<RESOURCE_NAME_LENGTH_MAX>         ResourceClassName;
```

7.4.9.3.3 Type **`ResourceGUID`**

This type is used to identify resource objects, see 7.4.2.3.1. It is defined as shown in the IDL below:

```
    typedef octet                ResourceGUID[16];
```

7.4.9.3.4 Type **`ResourcePathName`**

This type is used to identify resource objects using a human-readable representation, see 7.4.2.3.2. It is defined as shown in the IDL below:

```
    const int32 RESOURCE_NAME_LENGTH_MAX            = 255;
    typedef string<RESOURCE_NAME_LENGTH_MAX>         ResourceName;
```

7.4.9.3.5 Type **`ObservableElementId`**

This type is used to identify observable elements within the scope of the containing Resource Object, see 7.4.3.4.2. It is defined as shown in the IDL below:

```
    typedef uint32                                   ObservableElementId;
```

7.4.9.3.6 Type **`TypeObjectSerialized`**

This type is used to hold the full definition of a type. It is defined as shown in the IDL below:

```
    typedef sequence<octet>                TypeObjectSerialized;
```

The octet sequence shall contain the `CompleteTypeObject` as defined in sections 7.3.4.3 and 7.3.4.5 of DDS-XTYPES version 1.3, see [3], serialized as specified in section 7.3.4.5 of the aforementioned DDS-XTYPES specification.

7.4.9.3.7 Type **`TypeIdentifierSerialized`**

This type is used to uniquely identify a Type. It is defined as shown in the IDL below:

```
    typedef sequence<octet, 24>                TypeIdentifierSerialized;
```

This type shall contain the serialized representation of the TypeIdentifier as defined in section 7.3.4.2 of DDS-XTYPES version 1.3, see [3].
The Type Identifier shall be computed in the Complete Type Object, see XTYPES 7.3.4.3.
The serialization of the TypeIdentifier shall be done using XCDR version 3 with Little Endian encoding, see XTYPES 7.4.2.

7.4.9.3.8 Type **`TypeDefinition`**

This type is used to hold a TypeIndetifier alongside the full definition of a type. It is defined as shown in the IDL below:

```
    @appendable
```

```
struct TypeDefinition {
    TypeIdentifierSerialized          type_id;
    TypeObjectSerialized              type_object_serialized;
}
typedef sequence<TypeDefinition> TypeDefinitionSequence;
```

7.4.9.3.9 Type `ResourceMutableState` and `ResourceInmutableState`

This type is used to hold the state of a Resource alongside the `ResourceGUID` that identifies the resource.

The *ResourceInmutableState* is used to represent the state attributes that cannot change after the resource is created. The *ResourceMutableState* is used to represent and communicate changes to that state. The types are defined as shown in the IDL below:

```
@appendable
struct ResourceMutableState {
    sequence<ResourceGUID>          used_resources;
    sequence<ObservableElementId>   active_obsevable_element_ids;
};

@appendable
struct ResourceInmutableState {
    ResourceClassId                 class_id;
    ResourceName                    name;
    ResourceName                    namespace;
    ResourceGUID                    owner_resource;
    sequence<ResourceGUID>          required_resources;
    UserGUID                        user_guid;
};
```

The *ResourceMutableState* contains the following members:

- Member **used_resources**. Contains the list of resource objects currently associated with the resource identified by **guid**.
- Member **active_observable_element_ids**. Contains the list of Observable Elements in the resource that are being actively monitored.

The ResourceInmutableState contains the following members:

- Member **class_id**. Identifies the type of resource
- Member **name**. Holds the name of the resource.
- Member **owner_resource**. Applies to children resources. It identifies the parent resource.
- Member **user_guid**. Contains a user-provided identifier. It is not interpreted by DDS-Monitoring.

7.4.9.3.10 Types `Resource` and `ResourceReference`

The `Resource` type is used to hold the full definition of a Resource and its relationships to other Resources. It is defined as shown in the IDL below:

```
@final
struct Resource {
    ResourceGUID             guid;
    @observable_name("")
    ResourceMutableState     mutable_state;
    @observable_name("")
    ResourceInmutableState   inmutable_state;
};
typedef @external Resource           ResourceReference;
typedef sequence<ResourceReference>  ResourceList;
```

The `ResourceReference` type is used to hold a reference to a Resource.

7.4.9.3.11 Type `ResourceStateUpdate`

The `ResourcStateUpdate` type is used to communicate a change in the state of a Resource. It is defined as shown in the IDL below:

```
@final
struct ResourceStateUpdate {
    ResourceGUID             guid;
    @observable_name("")
    ResourceMutableState     mutable_state;
};
```

7.4.9.3.12 Type `ResourceList`

This type is used to hold the full description of a list of resources. It may serve multiple purposes:

- It may be used to communicate a snapshot of the existing resources, usually as a response to a query requesting the list of resources that meet some condition.
- It may be used to communicate the creation/addition of new resources to a system

The type is defined as shown in the IDL below:

```
typedef sequence<ResourceReference>         ResourceSequence;
```

7.4.9.3.13 ResourceClassId constants

For each Type in the Resource Model that has the `@resource` annotation, there shall be a constant of type `ResourceClassId` with name `<RESOURCECLASSNAME>_RESOURCE_CLASS_ID` where `<RESOURCECLASSNAME>` stands for the *ResourceClassName* (see 7.4.2.1.1) resource class name in upper case.
The value of the constant `<RESOURCECLASSNAME>_RESOURCE_CLASS_ID` shall be the *ResourceClassId* computed as specified in 7.4.2.1.3.

7.4.9.3.14 Type `EventInfo`

The `EventInfo` type is used to include additional data when sending Event information (see 7.6.8). It is defined as shown in the IDL below:

```
@appendable
struct EventInfo {
    ResourceGUID                root_resource_guid;
    uint64                      epoch_resource;
    uint64                      epoch_observable_unit;
    boolean                     is_snapshot;
};
```

The type contains the following members:

- Member **root_resource_guid**. The GUID of the root resource associated with the event. Note that the Event type (see 7.6.8). already contains the GUID of the resource that originated the Event (**resource_guid**). The **root_resource_guid** is the ancestor of the **resource_guid** which is the root resource in the resource tree (see 7.4.2.2).
- Member **epoch_resource**. Counts the number of changes that occurred in the **resource_guid** ResourceMutableState (see 7.4.9.3.9). This is used to detect missing updates to the resource state.
- Member **epoch_observable_unit**. Counts the number of changes that occurred in the *ObservableElement* that is causing the change Event notification to be sent. This is used to detect missing updates to the *ObservableElement*. Note that this observable element is also an *ObservableUnit*.
- Member **is_snapshot**. Indicates the Event Message is a response to a snapshot request sent via the Monitoring Administration Interface, see 7.7.3.5.

7.4.9.3.15 Type `RegistryPeriodic`

This type is used to send PERIODIC information about the Resource registry. In this version of the specification, it is defined as the empty type below:

```
        @mutable @nested @autoid
        struct RegistryPeriodic {
        };
```

7.4.9.3.16 Type `TypePeriodic`

This type is used to send PERIODIC information about the types in the system. In this version of the specification it is defined as the empty type below:

```
        @mutable @nested @autoid
        struct TypePeriodic {
        };
```

7.4.9.3.17 Type `RegistryEvent`

This type is used to send ON_CHANGE information about the Resource registry. It is defined as shown in the IDL below:

```
        @mutable @nested @autoid
        struct RegistryEvent {
            // Full list of resources in the registry
            @optional
            ResourceList             resource_snapshot;    /* first element root */
            @optional
            ResourceList             created_resources;
            @optional
            sequence<ResourceGUID>   deleted_resources;
            @optional
            ResourceStateUpdateList  updated_resources;
        };
```

This type contains the following members:

- Member **resource_snapshot**. This member contains a list of resources representing a snapshot of the registry. It is sent in response to a request received on the Administration interface (see 7.7). The list will contain the list of resources that match the request query.
- Member **created_resources**. This member contains a list of resources representing the resources that have been added to the registry. It is sent ON_CHANGE whenever resources are added.
- Member **deleted_resources**. This member contains a list of resource identifiers, representing the resources that have been deleted from the registry. It is sent ON_CHANGE whenever resources are deleted.
- Member **updated_resources**. This member contains a list of updates to existing resources. It is sent ON_CHANGE whenever resources are modified.

7.4.9.3.18 Type `TypeEvent`

This type is used to send ON_CHANGE information about the Resource registry. It is defined as shown in the IDL below:

```
        @mutable @nested @autoid
        struct TypeEvent {
            @optional
            sequence<TypeDefinition>   type_definitions;
            @optional
            TypeIdentifierSequence     dependent_type_ids;  /* Nested hash types */
        };
```

This type contains the following members:

- Member **type_descriptors**. This member contains a list of TypeDescriptos representing types in the system.
- .It is sent in response to a request received on the Administration interface (see 7.7). The list will contain the list of types that match the request query.

- Member **dependent_type_ids**. This member contains a list of TypeIdentifiers that are referenced by the types included in the **type_descriptors**. This information may be useful to the requester to identify other types it may need to also request.

7.4.9.3.19 Type `PeriodicUnionBase`

This type is used as a base type in the Monitoring Distribution Datamodel. It is defined as shown in the IDL below:

```
@appendable @nested
union EventUnionBase switch (ResourceClassId) {
  case REGISTRY_RESOURCE_CLASS_ID:
    RegistryEvent           registry;
  case TYPE_RESOURCE_CLASS_ID:
    TypeEvent               type;
};
```

7.4.9.3.20 Type `EventUnionBase`

This type is used as a base type in the Monitoring Distribution Datamodel. It is defined as shown in the IDL below:

```
@appendable @nested
union PeriodicUnionBase switch (ResourceClassId) {
  case REGISTRY_RESOURCE_CLASS_ID:
    RegistryPeriodic        registry;
  case TYPE_RESOURCE_CLASS_ID:
    TypePeriodic            type;
};
```

7.5 Distribution of the Monitoring Data

Monitoring Data may be gathered in two ways:

- Sampling a resource observable unit to get the data at some rate.
- By notification from the resource indicating a change has occurred to values in an observable unit.

The gathered *Monitoring Data* may be distributed also in two ways:

- PERIODICALLY: The *Monitoring Data* is sent at regular intervals, irrespective of any changes from the previously sent values for the same observable unit.
- ON_CHANGE: The *Monitoring Data* is sent only when (significant) changes occur in the values of a resource's observable unit.

This specification only focuses on the distribution of the *Monitoring Data* as it will be the aspect that impacts interoperability. The precise mechanism used to gather the *Monitoring Data* is considered implementation-specific. DDS Monitoring defines IDL annotations in the Resource Model that control how the monitoring data is distributed.

7.5.1 Periodic Data

For observable elements whose change is frequent and/or periodic, the direct approach is to gather data via "periodic sampling" also known as "polling". The monitoring infrastructure periodically polls (reads/queries) the value of the observable elements and generates the monitoring data from this. An example of this would be a metric recording the total number of bytes sent by an application since it started.

Note that when sampling is used, the values observed may be the same as previously seen. It is also possible that the observable elements had "intermediate" values missed by the sampling.

This approach requires that the infrastructure provides the query/read mechanisms to gather the observable data from resources.

Monitoring Data gathered periodically may also be distributed PERIODICALLY using the same period. However, in situations where the value changes are infrequent, or the semantics are such that "small" differences are not important, it may be more efficient to distribute them ON_CHANGE, that is compare the values with the previous ones distributed and send them only if the difference is deemed "significant". An example of this would be a metric recording the current bandwidth utilization in bytes per second. This metric may be gathered by polling internal counters periodically but distributed only if the utilization value differs from the previous more than a configured threshold.

As mentioned, DDS-Monitoring only concerns itself with the way the Monitoring Data is distributed, not how it is gathered.

The IDL `@observable` and `@observable_unit` annotations defined in 7.4.9.1.3, and 7.4.9.1.4 provide a way to configure which Observable Elements should be distributed PERIODICALLY.

7.5.2 On-Change Data

For observable elements whose change is infrequent and/or non-periodic the most natural (and likely efficient) approach is to capture the observable data "on change". In this approach, the monitoring infrastructure is notified each time the observable element changes at which point it can gather the Monitoring Data and make it available for distribution. An example of this would be a metric recording the total number of DDS DataReaders currently matched with a DataWriter.

The use of this approach requires that the infrastructure provides a change-notification mechanism that is sufficiently fine-grained to identify the resources and aspects of a resource that have changed. This is in addition to the query/read mechanisms to gather the observable data from resources upon receiving the change notification.

Monitoring Data gathered On-Change is typically distributed ON_CHANGE as well. However, it may be advantageous to not distribute every individual change separately. Rather the Monitoring infrastructure could accumulate changes for some time duration, trading off delay in the communication for efficiency.

The IDL `@observable` and `@observable_unit` annotations defined in 7.4.9.1.3, and 7.4.9.1.4 provide a way to configure which Observable Elements should be distributed ON_CHANGE.

7.5.3 Log Data

Log Messages are normally generated by the Software Infrastructure whenever relevant events occur. Typically, these log messages are printed to a console, written to a file, or sent to some system facility like syslog (available in Unix-like systems).

DDS Monitoring supports distributing the DDS Log Messages in addition to other Monitoring Data. This way the log messages become accessible to the same DDS Monitoring Infrastructure in a common way, regardless of the platform details where each DDS application is running.

Implementations of DDS Monitoring may offer implementation-specific mechanisms to configure logging in terms of verbosity level and enabled modules. Regardless of how it is configured, the Data Types and Topics used to send the Log information are specified in DDS Monitoring. That way applications sending or consuming the Log messages with interoperate independently of the DDS Monitoring implementation being used.

7.6 Monitoring Distribution Data Model

This specification uses OMG IDL to formally define the Monitoring Distribution Datamodel. This model is derived from the Monitoring Resource Model (see 7.4) by applying the rules in the subclauses below:

7.6.1 Use of IDL

The Monitoring distribution data model is defined using the OMG IDL language version 4.2.

7.6.2 Module Scope

All generated types shall appear in the module `monitoring::dds`.

7.6.3 Periodic structures for each Resource Type

For each (Structured) Type `<ResourceStructureName>` in the Resource Model there shall be a structure type called `<ResourceStructureName>Periodic`. Here `<ResourceStructureName>` stands for the name of the Structured type associated with the resource.

The structure type shall be constructed applying the following rules:

- The structure shall have the annotations `@mutable`, `@nested`, and `@autoid`.
- Every member of the structure shall have the `@optional` annotation
- The structure shall have at least one member for each descendent Observable Element (according to the Resource Observable Element Tree, see 7.4.3.3) of the resource `<ResourceStructureName>` that corresponds to an Observable Unit with `distributionKind=PERIODIC`, see 7.4.9.1.2 - 7.4.9.1.5. For each member:
    - The member type shall be the same as the type of the Observable Unit, we refer to this type as `<ResourceStructureMemberTypeName>`.
    - The member name shall be the *ElementPathSuffix* of the *ObservableElementName*, see 7.4.3.4.1. We refer to this member as `<resource_structure_member_name>`.
    - If the `<ResourceStructureMemberTypeName>` type associated with the member has multiple `@observable_view` annotations, then:
        - The structure shall include additional members of type `<ResourceStructureMemberTypeName>` in 1-to-1 correspondence with the views so that the number of members generated matches the number of view levels.
        - The member corresponding to the view with level **LI** =0 shall be named `<resource_structure_member_name>`. It shall also have the annotation `@view(level=0, member_level=0)`.
        - The members corresponding to views with level **LI** > 0 shall be named according to the pattern `<resource_structure_member_name>_LI`. These members shall have the annotation `@view(level=0, member_level=LI)`.
    - If the `<ResourceStructureMemberTypeName>` type associated with a member defines multiple serialization levels by having some members annotated with `@view`.
        - This case shall be treated as the previous case where the member Type has multiple `@observble_level` as these two notations are equivalent, see.

As an example, given the resource class name "simplified_application" with the declaration below:

```
@appendable @nested
@observable_unit
@observable_view(level=0, select="mean")
@observable_view(level=1, select="min;max")
@observable_view(level=2, select="%")
```

```
struct Int32Stat {
    uint32 period_ms;
    uint32 count;
    int32  mean;
    int32   min;
    int32  max;
};

struct NetworkUsage {
    @observable(distribution= PERIODIC) Int32Stat     messages_sent;
    @observable(distribution=ON_CHANGE)  Int32Stat     messages_received;
};

@mutable @nested
@resource(class="simplified_application", namespace="dds", owner="")
struct SimplifiedApplication   {
    @observable(distribution=ON_CHANGE) int32         cpu_temperature;
    @observable(distribution=PERIODIC)  int32         memory_usage;
    @observable                         NetworkUsage  network;
};
```

The `SimplifiedParticipantPeriodic` structure shall be defined as:

```
@mutable @nested @autoid
struct SimplifiedApplicationPeriodic {
    @optional                                  int32         memory_usage;
    @optional @view(level=0, member_level=0)  Int32Stat     network_messages_sent;
    @optional @view(level=0, member_level=1)  Int32Stat     network_messages_sent_1;
    @optional @view(level=0, member_level=2)  Int32Stat     network_messages_sent_2;
```

The member *messages_sent* results in 3 members each in `SimplifiedApplicationPeriodic`. This is because its type (`Int32Stat`) uses the `@observable_view` annotation to define 3 levels identified by the level values: 0, 1, and 2. The members *cpu_temparature* and *messages_received* do not appear in `SimplifiedApplicationPeriodic` because they do not have distributionKind PERIODIC.

7.6.4 Event structures for each Resource Type

For each (Structured) Type `<ResourceStructureName>` in the Resource Model that has been annotated as a resource there shall be a structure type called `<ResourceStructureName>Event`. Here `<ResourceStructureName>` stands for the name of the Structure Type associated with the resource.
The structure type shall be constructed applying the following rules:

- The structure shall have the annotations `@mutable`, `@nested`, and `@autoid`.
- Every member of the structure shall have the `@optional` annotation
- The structure shall have at least one member for each descendent Observable Element (according to the Resource Observable Element Tree, see 7.4.3.3) of the resource `<ResourceStructureName>` that corresponds to an Observable Unit with `distributionKind=ON_CHANGE`, see 7.4.9.1.2 - 7.4.9.1.5. For each member:
    - o The member type shall be the same as the type of the Observable Unit, we refer to this type as `<ResourceStructureMemberTypeName>`.
    - o The member name shall be the *ElementPathSuffix* of the *ObservableElementName*, see 7.4.3.4.1. We refer to this member as `<resource_structure_member_name>`.
    - o If the `<ResourceStructureMemberTypeName>` type associated with the member has multiple `@observable_view` annotations, then:

        - The structure shall include additional members of type `<ResourceStructureMemberTypeName>` in 1-to-1 correspondence with the views so that the number of members generated matches the number of view levels.
        - The member corresponding to the view with level **LI** =0 shall be named `<resource_structure_member_name>`. It shall also have the annotation `@view(level=0, member_level=0)`.
        - The members corresponding to views with level **LI** > 0 shall be named according to the pattern `<resource_structure_member_name>_LI`. These members shall have the annotation `@view(level=0, member_level=LI)`.
    - If the `<ResourceStructureMemberTypeName>` type associated with a member defines multiple serialization levels by having some members annotated with `@view`.
        - This case shall be treated as the previous case where the member Type has multiple `@observble_level` as these two notations are equivalent, see o.

As an example, given the resource class name "simplified_application" with the declaration below:

```
@appendable @nested
@observable_view(level=0, select="mean")
@observable_view(level=1, select="min;max")
@observable_view(level=2, select="%")
struct Int32Stat {
    uint32 period_ms;
    uint32 count;
    int32  mean;
    int32   min;
    int32  max;
};

struct NetworkUsage {
    @observable(distribution= PERIODIC) Int32Stat     messages_sent;
    @observable(distribution=ON_CHANGE)  Int32Stat     messages_received;
};

@mutable @nested
@resource(class="simplified_application", namespace="dds", owner="")
struct SimplifiedApplication   {
    @observable(distribution=ON_CHANGE) int32         cpu_temperature;
    @observable(distribution=PERIODIC)  int32         memory_usage;
    @observable                        NetworkUsage  network;
};
```

The `SimplifiedParticipantEvent` structure shall be defined as:

```
@extensibility(MUTABLE) @nested @autoid
struct SimplifiedApplicationEvent {
    @optional                             int32     cpu_temperature
    @optional @view(level=0, member_level=0)  Int32Stat network_messages_received;
    @optional @view(level=0, member_level=1)  Int32Stat network_messages_received_1;
    @optional @view(level=0, member_level=2)  Int32Stat network_messages_received_2;
```

The member *messages_received* results in 3 members each in `SimplifiedApplicationPeriodic`. This is because its type (`Int32Stat`) uses the `@observable_view` annotation to define 3 levels identified by the level values: 0, 1, and 2.

The members *memory_usage* and *messages_sent* do not appear in `SimplifiedApplicationPeriodic` because they do not have distributionKind ON_CHANGE.

7.6.5 PeriodicUnion type

There shall be a union type called `PeriodicUnion`. The type shall have the annotations `@appendable` and `@nested`.
The type shall be constructed applying the following rules:

- The `PeriodicUnion` shall extend the `PeriodicUnionBase`, see 7.4.9.3.19.
- The `PeriodicUnion` shall not have a default case.
- The `PeriodicUnion` shall have one case branch for each Type in the Resource Model that has the `@resource` annotation.
- The case discriminator values of the `PeriodicUnion` shall correspond to `<RESOURCECLASSNAME>_RESOURCE_CLASS_ID` constants defined in 7.4.9.3.13.
- The case member that corresponds to the case discriminator `<RESOURCECLASSNAME>_RESOURCE_CLASS_ID` shall have the type `<ResourceStructureName>Periodic` where `<ResourceStructureName>` is the Structure Type associated that declared the resource `<RESOURCECLASSNAME>`.

For example, assume the following resources:

```
@mutable  @nested
@resource(class="application", namespace="dds")
struct Application   { ... };

@mutable @nested
@resource(class="domain_participant", owner="application")
struct DomainParticipant   { ... };

@mutable @nested
@resource(class="topic", owner="domain_participant")
struct Topic   { ... };
```

The above resources would result in the following definition of the `PeriodicUnion`:

```
@appendable @nested
union PeriodicUnion : PeriodicUnionBase {
    case APPLICATION_RESOURCE_CLASS_ID:
      ApplicationPeriodic application;
    case DOMAIN_PARTICIPANT_RESOURCE_CLASS_ID:
      ParticipantPeriodic domain_participant;
    case TOPIC_RESOURCE_CLASS_ID:
      TopicPeriodic topic;
};
```

7.6.6 EventUnion type

There shall be a union type called `EventUnion`. The type shall have the annotations `@appendable` and `@nested`.
The type shall be constructed applying the following rules:

- The `EventUnion` shall extend the `EventUnionBase`, see 7.4.9.3.20
- The `EventUnion` shall not have a default case.
- The `EventUnion` shall have one case branch for each Type in the Resource Model that has the `@resource` annotation.
- The case discriminator values of the `EventUnion` shall correspond to `<RESOURCECLASSNAME>_RESOURCE_CLASS_ID` constants defined in 7.4.9.3.13.
- The case member that corresponds to the case discriminator `<RESOURCECLASSNAME>_RESOURCE_CLASS_ID` shall have the type

`<ResourceStructureName>Event` where `<ResourceStructureName>` is the Structure Type associated that declared the resource `<RESOURCECLASSNAME>`.

For example the resources:

```
@mutable @nested
@resource(class="application", namespace="dds")
struct Application   { ... };

@mutable @nested
@resource(class="domain_participant")
struct Participant   { ... };

@mutable @nested
@resource(class="topic")
struct Topic   { ... };
```

The above resources would result in the following definition of the `EventUnion`:

```
@appendable @nested
union EventUnion : EventUnionBase {
    case APPLICATION_RESOURCE_CLASS_ID:
      ApplicationEvent application;
    case DOMAIN_PARTICIPANT_RESOURCE_CLASS_ID:
      ParticipantEvent domain_participant;
    case TOPIC_RESOURCE_CLASS_ID:
      TopicEvent topic;
};
```

7.6.7 Periodic type (not nested)

There shall be a type called `Periodic`. The type shall have the definition shown in the IDL below.

```
@appendable @nested(false)
struct Periodic {
    GUID_t resource_guid;
    PeriodicUnion  value;
};
```

The type contains the following members:

- Member **resource_guid**. The GUID of the resource associated with the periodic update.
- Member **value**. The value of monitoring data associated with the resource for the most recent period (see 7.6.5).

7.6.8 Event type (not nested)

There shall be a type called `Event`. The type shall have the definition shown in the IDL below.

```
@appendable @nested(false)
struct Event {
    GUID_t resource_guid;
    @optional
    EventInfo   info;
    EventUnion  value;
};
```

The type contains the following members:

- Member **resource_guid**. The GUID of the resource associated with the event.
- Member **info**. Additional information related to the event, see 7.4.9.3.14.
- Member **value**. The value of monitoring data describing the event, see 7.6.6.

7.6.9 Logging type (not nested)

There shall be a type called `Logging`. The type shall have the definition shown in the IDL below.

```
@appendable
enum LoggingLevel {  // DDS-Security 1.5
    @value(0) EMERGENCY_LEVEL,     // System is unusable. Should not continue use.
    @value(1) ALERT_LEVEL, // Should be corrected immediately
    @value(2) CRITICAL_LEVEL,      // A failure in primary application.
    @value(3) ERROR_LEVEL, // General error conditions
    @value(4) WARNING_LEVEL,       // May indicate future error if action not taken.
    @value(5) NOTICE_LEVEL,// Unusual, but not erroneous event or condition.
    @value(6) INFORMATIONAL_LEVEL, // Normal operational. Requires no action.
    @value(7) DEBUG_LEVEL
};

@appendable
enum SyslogVerbosity {
    @value(0)   SILENT,
    @value(1)   EMERGENCY,
    @value(3)   ALERT,
    @value(7)   CRITICAL,
    @value(15)  ERROR,
    @value(31)  WARNING,
    @value(63)  NOTICE,
    @value(127) INFORMATIONAL,
    @value(255) DEBUG
};

const uint8 SYSLOG_FACILITY_USER    = 1;
const uint8 SYSLOG_FACILITY_SECURITY_EVENT = 10;
const uint8 SYSLOG_FACILITY_SERVICE= 22;
const uint8 SYSLOG_FACILITY_MIDDLEWARE      = 23;

@appendable @nested
struct LoggingSetting {
    SyslogVerbosity verbosity;
    uint8 facility;
};

@appendable @nested
@observable_unit(distribution=ON_CHANGE)
struct LoggingConfig {
    sequence<LoggingSetting> logging_collection;
    sequence<LoggingSetting> logging_forwarding;
};

@final @nested
struct NameValuePair {  // DDS-Security 1.5
    string name;
    string value;
};

@appendable @nested
struct LoggingMessage {  // DDS-Security 1.5
    uint8           facility;  // Set to 0x10. Indicates sec/auth msgs
    LoggingLevel    severity;
    Time_t          timestamp; // Since epoch 1970-01-01 00:00:00 +0000 (UTC)
    @optional string hostname;  // IP host name of originator
    @optional string hostip;    // IP address of originator
    @optional string appname;   // Identify the device or application
    @optional string procid;    // Process name/ID for syslog system
    string          msgid;     // Identify the type of message
    string          message;   // Free-form message
    // Note that certain string keys (SD-IDs) are reserved by IANA
    map<string, NameValuePairSeq>  structured_data;
    uint64           sn;        // Sequence number to uniquely identify msgs per
facility
};

@external typedef LoggingMessage LogReference;

@appendable @nested
struct LoggingEventInfo {
    GUID_t  root_resource_guid; // app guid
    uint64  epoch;
    boolean is_snapshot;
};


@appendable @nested(false)
struct Logging {
    LoggingEventInfo info;
    @optional sequence<LogReference> update;
    @optional sequence<LoggingMessage> snapshot;
};
```

The `Logging` type contains the following members:

- Member **info**. Information about the root resource associated with the Log message and total number of logs generated.
- Member **update**. Holder for a collection of Log messages
- Member **snapshot**. Holder for a collection of Log messages sent as a response to an administration request. See 7.7.3.6.

7.7 Monitoring Administration Datamodel

DDS Monitoring exposes a DDS-RPC Interface that allows remote applications to configure the resources and logs being distributed.

7.7.1 Use of IDL

The Monitoring Administration data model is defined using the OMG IDL language version 4.2. All the service operations are included in a single interface called `monitoring::Administration`.

The normative IDL definition of the `monitoring::Administration` interface is provided in the file `monitoring_administration.idl`, included in this specification.
The Administration interface shall be implemented using the remote procedure call protocol defined in the DDS-RPC specification [4]. Accordingly, the DDS Monitoring implementation shall create two Topics, one DataWriter and one DataReader, that are used to implement the `monitoring::Administration`:

- The DataReader `DDSMonitoringAdministrationRequestReader` shall subscribe to a Topic with name "DDSMonitoringAdministrationRequest" and Type `monitoring::AdministrationRequest`.
- The DataWriter `DDSMonitoringAdministrationReplyWriter` shall publish a Topic with name "DDSMonitoringAdministrationReply" and Type `monitoring::AdministrationReply`.

The types `monitoring::AdministrationRequest` and `monitoring::AdministrationReply` shall be derived from the `monitoring::Administration` interface in accordance with DDS-RPC.

7.7.2 Types used by the Monitoring Administration Interface

7.7.2.1 ResourcePathExpression

This type is used to hold an expression that may be used to match the `ResourcePathName` of one or more resources. See 7.4.2.3.2 for the definition of `ResourcePathName`.

The expression syntax and matching rules follow the POSIX `fnmatch` function [8]. In addition, the string "//" may be used to match a path prefix (with zero or more path elements) until a path element matching the pattern is found. This is similar to how it is used in the X-Path matching syntax.

It is defined in the IDL below.

```
typedef string ResourcePathExpression;
typedef sequence<ResourcePathExpression> ResourcePathExpressionSequence;
```

Examples:

```
    "/applications/myApp/domain_participants/myParticipant"
    "/applications/*/domain_participants/*/subscribers/*"
    "//subscribers/*"
```

The 3rd (last) pattern uses the matching expression "//" to match any path prefix the element /subscribers/ is found so it may be used instead of the second pattern.

7.7.2.2 ObservableElementNameExpression

This type is used to hold an expression that may be used to match the `ObservableElementName` of one or more resources. See 7.4.3.4.1 for the definition of `ObservableElementName`.

The expression syntax and matching rules follow the POSIX fnmatch function[8].

It is defined in the IDL below (all definitions are in the module `monitoring`).

```
typedef string ObservableElementNameExpression;
typedef sequence< ObservableElementNameExpression > ObservableElementNameExpressionSequence;
```

Examples:

```
    "dds_domain_participant/receive_samples_per_sec"
    "dds_domain_participant/send_* "
```

7.7.2.3 ObservableElementSelector

This type is used to select a set of `ObservableElements` (see 7.4.3). It is defined in the IDL below (all definitions are in the module `monitoring`).

```
@appendable @nested
struct ObservableElementSelector {
    ResourcePathExpression                          resource_path_selector;
    sequence< ObservableElementNameExpression >  observable_name_selectors;
};
typedef sequence<ObservableElementSelector> ObservableElementSelectorSequence;
```

The *resource_path_selector* is used to match a set of resources. For each of these resources, the matching algorithm iterates over all the `ObservableElementNameExpression` expressions in the *observable_name_selectors* sequence. Each expression is used to match against the `ObservableElementNames` in the Resource's Observable Element Tree.

If an `ObservableElementName` is matched by at least one of the expressions the corresponding *ObservableElement* will be selected.

7.7.2.4 ObservableElementChangeSet

This type is used to define a change on the list of `ObservableElements` that belong to a pre-existing set. The change is described in terms of a list of observable elements being added and removed.

It is defined in the IDL below (all definitions are in the module `monitoring`).

```
@appendable @nested
struct ObservableElementChangeSet {
    ResourcePathExpression                             resource_path_selector;
    sequence< ObservableElementNameExpression >  add_observable_name_selectors;
    sequence< ObservableElementNameExpression >  remove_observable_name_selectors;
};
typedef sequence< ObservableElementChangeSet > ObservableElementChangeSetSequence;
```

The *resource_path_selector* is used to match a set of resources, then for each of these resources:

- Each string in the *add_observable_selectors* contains an `ObservableElementNameExpression`. Each expression is used to match all the `ObservableElementNames` in the Resource. If an `ObservableElementName` is matched by at least one of the expressions the corresponding `ObservableElement` will be selected for addition.
- Each string in the *remove_observable_unit_selectors* contains an `ObservableElementNameExpression`. Each expression is used to match all the `ObservableElementNames` in the existing set. If an `ObservableElementName` is matched by at least one of the expressions the corresponding `ObservableElement` will be selected for removal.

7.7.2.5 LoggingVerbosityLevelSelector

This type is used to configure the log messages distributed by DDS Monitoring. It is defined in the IDL below (all definitions are in the module `monitoring`). See also 7.5.3 for the IDL definition of the types: `LoggingLevel`, `LoggingFacilityLevel`, and `SYSLOG_FACILITY` constants.

```
@appendable @nested
struct LoggingVerbosityLevelSelector {
    ResourcePathExpression                       application_path_selector;
    sequence<LoggingFacilityLevel>               logging_facility_level;
};
typedef sequence< LoggingVerbosityLevelSelector > LoggingVerbosityLevelSelectorSequence;
```

The *application_path_selector* is used to select a set of applications. This is done by matching the *ResourcePathNames* of resources that have Resource Class “application”.
For each of the selected applications, the *facility_verbosity_level* is used to specify the log verbosity level for each SysLog facility.

7.7.2.6 MonitoringCommandError

This type is used to return an exception from an operation in the monitoring::dds::Service interface. It is defined in the IDL below (all definitions are in the module `monitoring`).

```
@appendable
@nested
exception MonitoringCommandError {
    ReturnCode_t ret_code;
    string error_message;
};
```

7.7.3 Operations in the Monitoring Administration Interface

The interface is defined in the following IDL (all definitions are in the module `monitoring`).

```
@DDSService
@DDSRequestTopic (name="DDSMonitoringAdministrationRequest")
@DDSReplyTopic   (name=" DDSMonitoringAdministrationReply")
interface Administration {

void set_subscription_state(
        in sequence<ObservableElementSelector> observable_element_selectors)
        raises (MonitoringCommandError);

void update_subscription_state(
        in sequence<ObservableElementChangeSet> observable_element_change_sets)
        raises (MonitoringCommandError);

void set_logging_verbosity(
        in sequence<LoggingVerbosityLevelSelector> logging_verbosity_level_selectors)
        raises (MonitoringCommandError);

void request_resource_registry(
        in sequence<ResourcePathExpression>  application_selector)
        raises (MonitoringCommandError);

void request_observable_element_snapshot(
        in sequence<ObservableElementSelector>  observable_element_selectors)
        raises (MonitoringCommandError);

void request_logging_snapshot(
        in sequence<LoggingVerbosityLevelSelector> logging_verbosity_level_selectors)
        raises (MonitoringCommandError);

void request_type_definitions(
        in sequence<TypeIdentifierSerialized> type_ids,
        in boolean include_required_type_ids)
        raises (MonitoringCommandError);
};
```

7.7.3.1 Operation: set_subscription_state

Configures the complete set of `ObservableElements` that the DDS Monitoring infrastructure should include in the Monitoring Data it sends to the requesting application.

The operation has the following parameters:

- Parameter **observable_element_selectors**. Used to specify the `ObservableElements` that are of interest. Each element of the sequence is an `ObservableElementSelector` that matches a collection of `ObservableElements`. The union of all these matches becomes the "subscribed" set of `ObservableElements`. This means:
    - Any previously-subscribed `ObservableElements` that are not matched by the **observable_element_selectors** become unsubscribed.
    - Any previously un-subscribed `ObservableElements` that are matched by the **observable_element_selectors** become subscribed.
- Parameter **exception.** Used to report errors in the processing of the request command.

7.7.3.2 Operation: update_subscription_state

Modifies the set of `ObservableElements` that the DDS Monitoring infrastructure should include in the Monitoring Data it sends to the requesting application.

Unlike `set_subscription_state`, this operation specifies a delta change, meaning it specifies `ObservableElements` to add and remove from the currently subscribed set.

The operation has the following parameters:

- Parameter **observable_unit_change_sets**. Used to specify the `ObservableElements` that are added and removed. Each element of the sequence is an `ObservableElementChangeSet` that specifies a list of `ObservableElements` to add and remove.
    - Starting from the set of `ObservableElements` currently subscribed, each of these additions and removals is performed resulting in an updated set of subscribed `ObservableElements`.
- Parameter **exception.** Used to report errors in the processing of the request command.

7.7.3.3 Operation: set_logging_subscription_state

Configures the Log messages that the DDS Monitoring infrastructure should include in the Monitoring Data it sends to the requesting application.

The operation has the following parameters:

- Parameter **logging_verbosity_level_selectors**. Used to specify the Log Messages that are of interest. Each element of the sequence is a `LoggingVerbosityLevelSelector` that specifies the verbosity level for each Log category on a set of application resources.
- Parameter **exception.** Used to report errors in the processing of the request command.

7.7.3.4 Operation: request_resource_registry

Requests the DDS Monitoring infrastructure to send the resources contained by a set of applications.

The operation has the following parameters:

- Parameter **application_selector**. Used to match against the ResourcePathNames corresponding to application resources. Any application resource that is matched by one of the expressions in the **application_selector** becomes a target for the request.
- Parameter **exception.** Used to report errors in the processing of the request command.

7.7.3.5 Operation: request_observable_element_snapshot

Requests the DDS Monitoring infrastructure to send a one-time copy of the latest Monitoring Data corresponding to the specified `ObservableElements`.

Unlike `set_subscription_state` and `update_subscription_state`, this operation requests DDS Monitoring to send the data just once. It does not request the Monitoring infrastructure to send future changes to the values of the `ObservableElements`.

The operation has the following parameters:

- Parameter **observable_element_selectors**. Used to specify the `ObservableElements` that are of interest. Each element of the sequence is an `ObservableElementSelector` that matches a collection of `ObservableElements`. The union of all these matches becomes the set of `ObservableElements` that are requested in the snapshot.
- Parameter **exception.** Used to report errors in the processing of the request command.

7.7.3.6 Operation: request_logging_snapshot

Requests the DDS Monitoring infrastructure to send a one-time copy of the Log Messages that match the request specification.

The operation has the following parameters:

- Parameter **logging_verbosity_level_selectors**. Used to specify the Log Messages that are of interest. Each element of the sequence is a `LoggingVerbosityLevelSelector` that specifies the verbosity level for each Log category on a set of application resources.
- Parameter **exception.** Used to report errors in the processing of the request command.

7.7.3.7 Operation: request_type_definitions

Requests the DDS Monitoring infrastructure to send a one-time copy of the Type Definitions that match the request specification.

The operation has the following parameters:

- Parameter **type_ids**. Used to specify the TypeIdentifiers for the requested types
- Parameter **include_required_type_ids**. Used to specify that the response should include a list of the TypeIdentifiers that the included definitions depend on. This can help minimize the number of RPC calls to retrieve a type alongside all the types it depends on. TypeIdentiofiers for the requested types

Parameter **exception.** Used to report errors in the processing of the

7.8 Monitoring Distribution Protocol

Monitoring Data is distributed to external client applications using a communication protocol. This can be described in terms of a middleware platform and corresponding data models and services used with the platform.

7.8.1 Middleware Platform

This specification uses the OMG Data-Distribution Service (DDS) [1] and the DDS Real-Time Publish-Subscribe Protocol (DDS-RTPS) [2] as the Middleware platform to distribute the monitoring data to external consumers.
The monitoring data is represented using the Monitoring Distribution Data Model (see 7.6), when the monitoring data is sent on the network via DDS it shall be serialized according to the rules defined in the DDS-XTYPES version 1.3 specification [3] applied to the types in the IDL-specified data model.
DDS-Security [5] may be used to provide access control to the Monitoring Data and ensure the confidentiality and integrity of the Monitoring Data.

7.8.2 Middleware Services

The services built on DDS used to distribute the *Monitoring Data* are fully specified in terms of:

- The collection of DDS Topics and associated Data Types (aligned with the distribution data model) used to send and receive the data
- The DDS quality of service (QoS) of the DataWriters and DataReaders used to publish/subscribe the Topics.
- A collection of DDS-RPC service interfaces and associated data types used to remotely configure the service.

7.8.2.1 DDS Topics and Types

A specific configuration of DDS Monitoring will distribute the data with the types described in the Monitoring Distribution Model which is derived from a corresponding Monitoring Resource Model (see 7.8).

The Monitoring Distribution Model has only three non-nested (top-level) types: the IDL-defined structures `Periodic`, `Event`, and `Logging`, see 7.6.7, 7.6.8, and . These types are used to send the Monitoring Data for all Resources.

The DDS Monitoring Distribution shall use the following three Topics to send the Monitoring Data.

- The "DDSMonitoringPeriodic" Distribution Topic is used to send the Monitoring Data resulting from the observable elements configured with distribution kind PERIODIC. The associated data type is the `Periodic` structure defined in 7.6.7.
- The "DDSMonitoringEvent" Distribution Topic is used to send the Monitoring Data resulting from the observable elements configured with distribution kind ON_CHANGE. The associated data type is the `Event` structure defined in 7.6.7. This topic is also used to send the Resource Tree.
- The "DDSMonitoringLogging" Distribution Topic is used to send the Logging Data resulting from the Log Messages emitted by the middleware infrastructure.

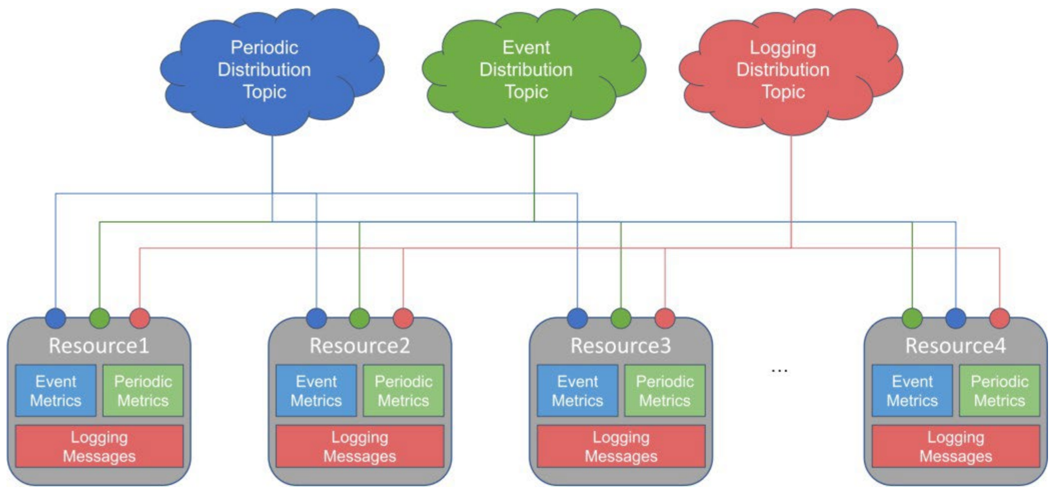


Figure 8.3: Resource distribution mapping into DDS Distribution Topics

The DDS Monitoring Distribution Layer shall create three DataWriters to send Monitoring Data: `DDSMonitoringPeriodicWriter`, `DDSMonitoringEventWriter`, and `DDSMonitoringLoggingWriter`.

- `DDSMonitoringPeriodicWriter` shall publish the topic name "DDSMonitoringPeriodic" with type `monitoring::dds::Periodic`.
- `DDSMonitoringEventWriter` shall publish the topic name "DDSMonitoringEvent" with type `monitoring::dds::Event`.
- `DDSMonitoringLoggingWriter` shall publish the topic name "DDSMonitoringLogging" with type `monitoring::dds::Logging`.

These DataWriter entities shall have the Qos defined in 7.8.2.3.

7.8.2.2 DDS-RPC Services

The DDS Monitoring Distribution Layer shall implement the `monitoring::Administration` interface as specified in the DDS-RPC specification. It shall create two Topics "DDSMonitoringAdministrationRequest" and "DDSMonitoringAdministrationReply", one DataReader and one DataWriter:

- The DataReader `DDSMonitoringAdministrationRequestReader` shall subscribe to the Topic "DDSMonitoringAdministrationRequest" with Type `monitoring::AdministrationRequest`.
- The DataWriter `DDSMonitoringAdministrationReplyWriter` shall publish the Topic "DDSMonitoringAdministrationReply" with Type `monitoring::AdministrationReply`.

The types `monitoring::AdministrationRequest` and `monitoring::AdministrationReply` shall be derived from the `monitoring::Administration` interface as specified by DDS-RPC. These types are specified in the `monitoring_administration.idl` which is included as part of the specification.

These DataWriter and DataWriter entities shall have the Qos defined in 7.8.2.3.

7.8.2.3 DDS QoS

The `DDSMonitoringPeriodicWriter`, `DDSMonitoringEventWriter`, and `DDSMonitoringLoggingWriter` have a fixed set of QoS settings that are adequate for the nature of the information flow they carry.

The `DDSMonitoringAdministrationRequestReader` and `DDSMonitoringAdministrationReplyWriter` also have a fixed set of QoS settings which are specified in DDS-RPC as default QoS for the DDS Entities implementing the Service request and reply.

Table 8.1 specifies the differentiating QoS settings for each of them. Qos Policies that are not specified shall be set to their corresponding default values.

Table 8.1: QoS settings for the DDS Monitoring DataWriters and DataReaders

	DDS QoS Policy		
DDS Monitoring Endpoint	**RELIABILITY kind**	**DURABILITY kind**	**HISTORY kind**
`DDSMonitoringPeriodicWriter`	BEST_EFFORT	VOLATILE	KEEP_ALL
`DDSMonitoringEventWriter`	RELIABLE	VOLATILE	KEEP_ALL
`DDSMonitoringLoggingWriter`	RELIABLE	VOLATILE	KEEP_ALL
`DDSMonitoringAdministrationRequestReader`	RELIABLE	VOLATILE	KEEP_ALL
`DDSMonitoringAdministrationReplyWriter`	RELIABLE	VOLATILE	KEEP_ALL

7.8.3 Security

DDS Monitoring uses DDS to distribute the Monitoring Data and send the administration commands that configure the Monitoring infrastructure. Applications that have a security requirement for their monitoring data shall use DDS Security to protect the DDS Topics and Services used by DDS Monitoring.

As described in 7.8.2.1 and 7.8.2.2, DDS Monitoring creates 4 DataWriters and 1 DataReader:

- DDSMonitoringPeriodicWriter with Topic "DDSMonitoringPeriodic"
- DDSMonitoringEventWriter with Topic "DDSMonitoringEvent"
- DDSMonitoringLoggingWriter with Topic "DDSMonitoringLogging"
- DDSMonitoringAdministrationRequestReader with Topic "DDSMonitoringAdministrationRequest"
- DDSMonitoringAdministrationReplyWriter with Topic " DDSMonitoringAdministrationReply"

In accordance with DDS-Security, to protect the system:

- These Topic names should appear in the shared Governance file so the security deployment requirements for these Topics can be specified.
    - Alternatively, the governance file may use Topic-Name expressions that cover these topic names.
- These Topic names should appear in the Permissions file of every application that is being monitored, the permission file should contain grants allowing the "DDSMonitoringAdministrationRequest" to be read and the remaining Topics to be written.
    - Alternatively, the permissions file may use Topic-Name expressions that cover these topic names.

8 Full IDL definition of DDS Monitoring Data models

8.1 DDS Monitoring Resource Model

The full definition of the DDS Monitoring Resource model is provided in the following machine-readable files which are included in this specification:

- `monitoring_annotations.idl`
- `monitoring_resource.idl`
- `monitoring_statistics.idl`
- `monitoring_logging.idl`
- `monitoring_administration.idl`
- `monitoring_dds_common.idl`
- `monitoring_dds_status.idl`
- `monitoring_dds_qos.idl`
- `monitoring_dds_entities.idl`

8.2 DDS Monitoring Distribution Model

The DDS Monitoring Distribution Model is obtained by applying the rules in clause 7.6 to the types in the DDS Monitoring Resource Model (see 8.1).

The DDS Monitoring Distribution Model resulting from the application of these rules is provided in the machine-readable files `monitoring_dds_distribution.idl` and `monitoring_dds_distribution_constants.idl` which are included in this specification.

The file `monitoring_dds_distribution.idl` includes the IDL files in the monitoring resource model.

Annex A – 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 2.5, http://www.omg.org/spec/DDSI-RTPS/

[3] DDS-XTYPES: Extensible and Dynamic Topic-Types for DDS version 1.3 http://www.omg.org/spec/DDS-XTypes/

[4] DDS-RPC: RPC Over DDS version 1.0. https://www.omg.org/spec/DDS-RPC/

[5] DDS-Security: DDS Security version 1.2. https://www.omg.org/spec/DDS-SECURITY/

[6] OMG-IDL: Interface Definition Language (IDL) version 4.2 http://www.omg.org/spec/IDL/

[7] The Syslog Protocol. IETF RFC 5424. https://www.rfc-editor.org/rfc/rfc5424.txt

[8] POSIX fnmatch. The Open Group Base Specifications Issue 7, 2018 fnmatch function. https://pubs.opengroup.org/onlinepubs/9699919799/functions/fnmatch.html