Unmanned Maritime Autonomy Architecture (UMAA)
Support Operations (SO)
Interface Control Document (ICD)
(UMAA-SPEC-SOICD)

MDE Version 5.0.1

UMAA Spec Commit 63fb9f9

Version 5.2.1
20 April 2022

DISTRIBUTION STATEMENT A: Approved for public release. Distribution is unlimited.

Contents

List of Figures

List of Tables

1 Scope

1.1 Identification

This document defines a set of services and interfaces as part of the Unmanned Maritime Autonomy Architecture (UMAA). The services and corresponding interfaces covered in this ICD encompass the functionality to provide support operations for an Unmanned Maritime Vehicle (UMV) (surface or undersea). As such, it provides services used across functional boundaries of UMAA ICD's such as logging, supporting startup and shutdown, providing emissions services, and resource control. This document is generated automatically from data models that define its services and interfaces as part of the Unmanned Systems (UxS) Control Segment (UCS) Architecture as extended by UMAA to provide autonomy services for unmanned vehicles.

To put each ICD in context of the UMAA Architecture Design Description (ADD), the UMAA functional decomposition mapping to UMAA ICDs is shown in Figure 1.

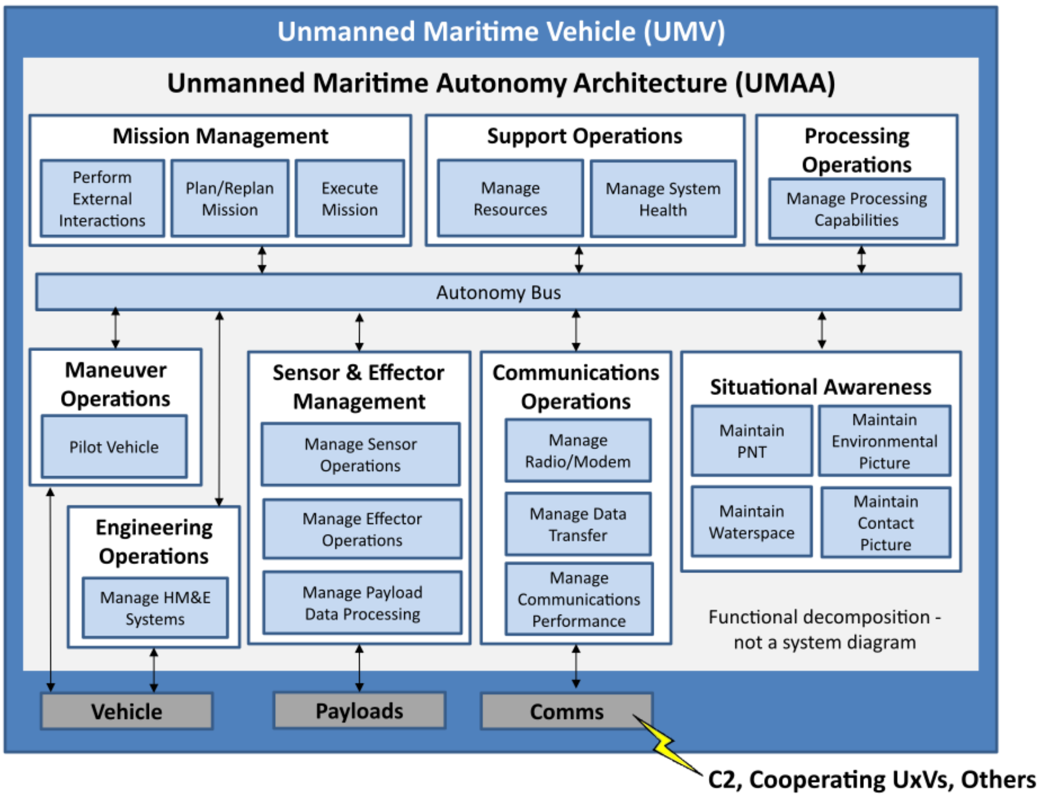

Figure 1: UMAA Functional Organization.

1.2 Overview

The fundamental purpose of UMAA is to promote the development of common, modular, and scalable software for unmanned vehicles that is independent of a particular autonomy implementation. Unmanned Maritime Systems (UMSs) consist of Command and Control (C2), one or more unmanned vehicles, and support equipment and software (e.g. recovery system, Post Mission Analysis applications). The scope of UMAA is focused on the autonomy that resides on-board the unmanned vehicle. This includes the autonomy for all classes of unmanned vehicles and must support varying levels of communication in mission (i.e., constant, intermittent, or none) with external systems. To enable modular development and upgrade of the functional capabilities of the on-board autonomy, UMAA defines eight high-level functions. These core functions include: Communications Operations, Engineering Operations, Maneuver Operations, Mission Management, Processing Operations, Sensor and Effector Operations, Situational Awareness, and Support Operations. In each of these areas, it is anticipated that new capabilities will be required to satisfy evolving Navy missions over time. UMAA seeks to define standard interfaces for these functions so that individual programs can leverage capabilities developed to these standard interfaces across programs that meet the standard interface specifications. Individual programs may group services and interfaces into components in different ways to serve their particular vehicle's needs. However, the entire interface defined by UMAA will be required as defined in the ICDs for all services that are included in a component. This requirement is what enables autonomy software to be ported between heterogeneous UMAA-compliant vehicles with their disparate vendor-defined vehicle control interfaces without recoding to a vehicle-specific interface.

Support Operations provides capabilities for services that are shared across all of the other functional areas within UMAA. This support includes the ancillary infrastructure services and interfaces required to operate an unmanned vehicle. Standard interfaces are defined for startup and shutdown, logging of time-stamped event and attribute data, operational mode control (e.g. operational, simulation, maintenance, training), and resource control (i.e. managing which client is in control of a component).

Unlike the primary concerns of an unmanned vehicle system, such as propulsion control and sensor data processing, the support operations are not typically seen in an external view of the system. Standardization of these services provides a consistent way to manage internal modes and control hierarchies across platforms and programs.

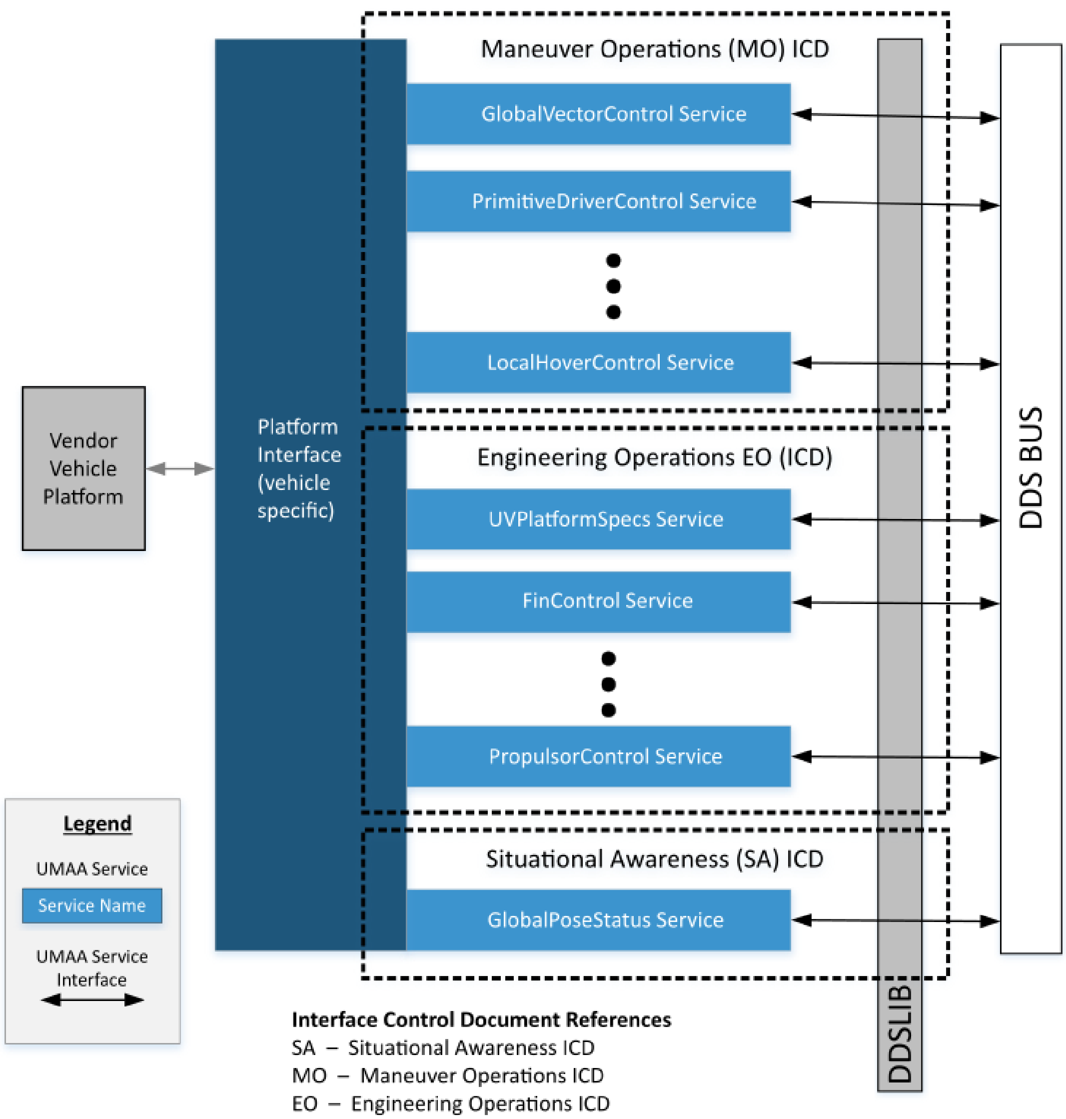

Figure 2: UMAA Services and Interfaces Example.

1.3 Document Organization

This interface control document is organized as follows:

Section 1 – Scope: A brief purview of this document

Section 2 – Referenced Documents: A listing of associated of government and non-government documents and standards

Section 3 – Introduction to Data Model, Services, and Interfaces: A description of the common data model across all services and interfaces

Section 4 – Flow Control: A description of different flow control patterns used throughout UMAA

Section 5 – Support Operations (SO) Services and Interfaces: A description of specific services and interfaces for this ICD

2 Referenced Documents

The documents in the following table were used in the creation of the UMAA interface design documents. Not all references may be applicable to this particular document.

Table 1: Standards Documents

Title	Release Date
A Universally Unique IDentifier (UUID) URN Namespace	July 2005
Data Distribution Service for Real-Time Systems Specification, Version 1.4	March 2015
Data Distribution Service Interoperability Wire Protocol (DDSI-RTPS), Version 2.3	April 2019
Object Management Group Interface Definition Language Specification (IDL)	March 2018
Extensible and Dynamic Topic Types for DDS, Version 1.3	February 2020
UAS Control Segment (UCS) Architecture, Architecture Description, Version 2.4	27 March 2015
UCS Architecture, Conformance Specification, Version 2.2	27 September 2014
UCS-SPEC-MODEL v3.4 Enterprise Architect Model	27 March 2015
UCS Architecture, Architecture Technical Governance, Version 2.5	27 March 2015
System Modeling Language Specification, Version 1.5	May 2017
Unified Modeling Language Specification, Version 2.5.1	December 2017
Interface Definition Language (IDL), Version 4.2	March 2018
U.S. Department Of Homeland Security, United States Coast Guard "Navigation Rules International-Inland" COMDTINST M16672.2D	March 1999
IEEE 1003.1-2017 - IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7	December 2017
Guard, U. C. (2018). Navigation Rules and Regulations Handbook: International—Inland. Simon and Schuster.	June 2018
Department of Defense Interface Standard: Joint Military Symbology (MIL-STD-2525D Appendix A)	10 June 2014
DOD Dictionary of Military and Associated Terms	August 2018

Table 2: Government Documents

Title	Release Date
Unmanned Maritime Autonomy Architecture (UMAA) Architecture Design Description (ADD), Version 1.0	January 2019
Manual for the Submission of Oceanographic Data Collected by Unmanned Undersea Vehicles (UUVs)	October 2018

3 Introduction to Data Model, Services, and Interfaces

3.1 Data Model

A common data model is at the heart of UMAA. The common data model describes the entities that represent system state data, the attributes of those entities and relationships between those entities. This is a "data at rest" view of system-level information. It also contains data classes that define types of messages that will be produced by components, or a "data in motion" view of system-level information.

The common data model and coordinated service interfaces are described in a Unified Modeling Language (UML™) modeling tool and are represented as UML™ class diagrams. Interface definition source code for messages/topics and other interface definition products and documentation will be automatically generated from the common data model so that they are consistent with the data model and to ensure that delivered software matches its interface specification.

The data model is maintained as a Multi-Domain Extension (MDE) to the UCS Architecture and will be maintained under configuration control by the UMAA Board as UCSMDE and will be incrementally integrated into the core UCS standard. Section 5 content is automatically generated from this data model, as are other automated products such as IDL that are used for automated code generation.

3.2 Definitions

UMAA ICDs follow the UCS terminology definitions found in the UCS Architecture Description v2.4. The normative (required) implementation to satisfy the requirements of a UMAA ICD is to provide service and interface specification compliance. Components may group services and required interfaces in any manner so long as every service meets its interface specifications. Figure 3 shows a particular grouping of services into components. The interfaces are represented by the blue and green lines and may equate to one or more independent input and output interfaces for each service. The implementation of the service into software components is left up to the individual system development. Given this context, section 5 correspondingly defines services with their interfaces and not components.

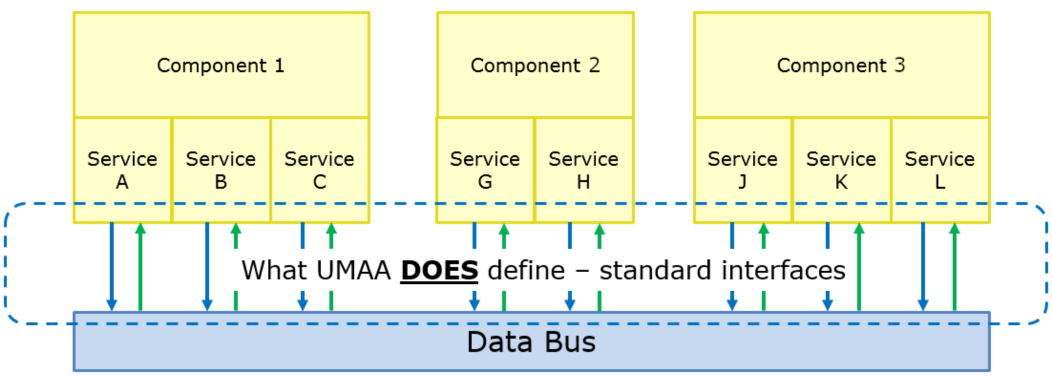

Figure 3: Services and Interfaces Exposed on the UMAA Data Bus.

Services may use other services within this ICD, or in other UMAA defined ICDs, to provide their capability. Additionally, components for acquisition and development may span multiple ICDs. An example of this would be a commercial radar that provides both status and control of the unit via the radar's software Application Programming Interface (API).

3.3 Data Distribution Service (DDS™)

The data bus supporting autonomy messaging (as seen in Figure 3) is implemented via DDS™. DDS is a middleware protocol and API standard for data-centric connectivity from the Object Management Group (OMG). It integrates the components of a system together, providing low-latency data connectivity, extreme reliability, and a scalable architecture. In a distributed system, middleware is the software layer that lies between the operating system and applications. It enables the various system components to more easily communicate and share data. It simplifies the development of distributed systems by letting software developers focus on the specific purpose of their applications rather than the mechanics of passing information between applications and systems. The DDS specification is fully described in free reference material on the OMG website and there are both open source and commercially available implementations.

3.4 Naming Conventions

UMAA services are modeled within the UCS Architecture under the Multi-Domain Extension (MDE). The UCS Architecture uses SoaML concepts of participant, serviceInterface, service port, and request port to describe the interfaces that make up a service and show how the service is used. Each service defines the capability it provides as well as required interfaces. Each interface consists of an operation that accepts a single message (A SoaML MessageType). In SoaML, a MessageType is defined as a unit of information exchanged between participant Request and Service ports via ServiceInterfaces. Instances of a MessageType are passed as parameters in ServiceInterface operations. (Reference: UCS Architecture, Architecture Technical Governance)

To promote commonality across service definitions, a common way of naming services and their sets of operations and messages has been adopted for defining services within UCS-MDE. The convention uses the Service Base Name <SBN> and an optional Function Name [FN] to derive all service names and their associated operations and messages. As this is meant to be a guide, services might not include all of the defined operations and messages and their names might not follow the convention where a more appropriate name adds clarity.

Furthermore, services in UMAA are not required to be defined as indicated in Table 3 when all parts of the service capabilities are required for the service to be meaningful (such as ResourceAllocation).

Additionally, note that for UMAA not all operations defined in UCS-MDE result in a message being published to the DDS bus, e.g., since DDS uses publish/subscribe, most query operations result in a subscription to a topic and do not actually publish the associated request message. In the case of cancel commands, there is no associated implementation of the cancel<SBN>[FN]CommandStatus as it is just the intrinsic response of the DDS dispose function; so, it is essentially a NOOP (no operation) in implementation. The conventions used to define UCS-MDE services are as follows:

Service Name
<SBN>[FN]Config
<SBN>[FN]Control
<SBN>[FN]Specs
<SBN>[FN]Status OR Report

where the SBN should be descriptive of the task or information provided by the service. Note that the FN is optional and only included if needed to clarify the function of the service. The suffixes Status and Report are interchangeable. If a "Report" is a more appropriate description of the service, it can be used in lieu of "Status".

Table 3: Service Requests and Associated Responses

	Service Requests (Inputs)	**Service Responses (Outputs)**
Config	set<SBN>[FN]Config	report<SBN>[FN]ConfigCommandStatus
	query<SBN>[FN]ConfigAck	report<SBN>[FN]ConfigAck
	query<SBN>[FN]Config	report<SBN>[FN]Config
	cancel<SBN>[FN]Config	report<SBN>[FN]CancelConfigCommandStatus
	query<SBN>[FN]ConfigExecutionStatus	report<SBN>[FN]ConfigExecutionStatus
Control	set<SBN>[FN]	report<SBN>[FN]CommandStatus
	query<SBN>[FN]CommandAck	report<SBN>[FN]CommandAck
	cancel<SBN>[FN]Command	report<SBN>[FN]CancelCommandStatus
	query<SBN>[FN]ExecutionStatus	report<SBN>[FN]ExecutionStatus
Specs	query<SBN>[FN]Specs	report<SBN>[FN]Specs
Status OR Report	query<SBN>[FN]	report<SBN>[FN]

Service Requests (operation:message)
set<SBN>[FN]Config:<SBN>[FN]ConfigCommandType

query<SBN>[FN]Config:<SBN>[FN]ConfigRequestType[1]
set<SBN>[FN]:<SBN>[FN]CommandType
query<SBN>[FN]CommandAck:<SBN>[FN]CommandAckRequestType[1]
cancel<SBN>[FN]Command:<SBN>[FN]CancelCommandType[1]
cancel<SBN>[FN]Config:<SBN>[FN]CancelConfigType[1]
query<SBN>[FN]ExecutionStatus:<SBN>[FN]ExecutionStatusRequestType[1]
query<SBN>[FN]ConfigExecutionStatus:<SBN>[FN]ConfigExecutionStatusRequestType[1]
query<SBN>[FN]ConfigAck:<SBN>[FN]ConfigAckRequestType[1]
query<SBN>[FN]Specs:<SBN>[FN]SpecsRequestType[1]
query<SBN>[FN]:<SBN>[FN]RequestType [1] [2]

Service Responses (operation:message)
report<SBN>[FN]ConfigCommandStatus:<SBN>[FN]ConfigCommandStatusType
report<SBN>[FN]Config:<SBN>[FN]ConfigReportType
report<SBN>[FN]ConfigAck:<SBN>[FN]ConfigAckReportType
report<SBN>[FN]CommandStatus:<SBN>[FN]CommandStatusType
report<SBN>[FN]CommandAck:<SBN>[FN]CommandAckReportType
report<SBN>[FN]CancelCommandStatus:<SBN>[FN]CancelCommandStatusType[1]
report<SBN>[FN]CancelConfigCommandStatus:<SBN>[FN]CancelConfigCommandStatusType[1]
report<SBN>[FN]ExecutionStatus:<SBN>[FN]ExecutionStatusReportType
report<SBN>[FN]ConfigExecutionStatus:<SBN>[FN]ConfigExecutionStatusReportType
report<SBN>[FN]Specs:<SBN>[FN]SpecsReportType
report<SBN>[FN]:<SBN>[FN]ReportType

where,

- Config (Configuration) Command/Report – This is the setup of a resource for operation of a particular task. Attributes may be static or variable. Examples include: maximum RPM allowed, operational sonar frequency range allowed, and maximum allowable radio transmit power.
- Command Status – This is the current state of a particular command (either control or configuration).
- Command – This is the ability to influence or direct the behavior of a resource during operation of a particular task. Attributes are variable. Examples include a vehicle's speed, engine RPM, antenna raising/lowering, and controlling a light or gong.
- Command Ack (Acknowledgement) Report – This is the command currently being executed.
- Cancel – This is the ability to cancel a particular command that has been issued.
- Execution Status Report – This is the status related to executing a particular command. Examples associated with a waypoint command include cross track error, time to achieve, and distance remaining.
- Specs (Specifications) Report – Provides a detailed description of a resource and/or its capabilities and constraints. Attributes are static. Examples include: maximum RPM of a motor, minimum frequency of a passive sonar sensor, length of the unmanned vehicle, and cycle time of a radar.
- Report – This is the current information being provided by a resource. Examples include vehicle speed, rudder angle, current waypoint, and contact bearing.

3.5 Namespace Conventions

Each UMAA service and the messages under the service can be accessed through their appropriate UMAA namespace. The namespace reflects the mapping of a specific service to its parent ICD, and the parent ICD's mapping to the overall UMAA Design Description. For example:

Access the Primitive Driver Control service under Maneuver Operations:
UMAA::MO::PrimitiveDriverControl
Access the ContactReport Service under Situational Awareness:

[1] These message types are required for UCS model rules of construction, but are not implemented as messages in the UMAA specification.
[2] At this time, there are no Requests in the specification. This will be the message format when Requests have been added.

UMAA::SA::ContactReport

The UMAA model uses common data types that are re-used through the model to define service interface topics, interface topics, and other common data topics. These data types are not intended to be directly utilized but, for reference, they can be accessed in the same manner:

Access the common UMAA Status Message Fields:
UMAA::UMAAStatus
Access the common UMAA GeoPosition2D (i.e., latitude and longitude) structure:
UMAA::Common::Measurement::GeoPosition2D

3.6 Cybersecurity

The UMAA standard addressed in this ICD is independent from defining specific measures to achieve Cybersecurity compliance. This UMAA ICD does not preclude the incorporation of security measures, nor does it imply or guarantee any level of Cybersecurity within a system. Cybersecurity compliance will be performed on a program-specific basis and compliance testing is outside the scope of UMAA.

3.7 GUID algorithm

The UMAA standard utilizes the Globally Unique IDentifier (GUID), conforming to the variant defined in RFC 4122 (variant value of 2). Generators of GUIDs may generate GUIDs of any valid, RFC 4122-defined version that is appropriate for their specific use case and requirements. (Reference: A Universally Unique IDentifier (UUID) URN Namespace)

3.8 Large Collections

The UMAA standard defines Large Collections, which are collections of decoupled but related data. Large Collections provide the ability to update one or more elements of the collection without republishing the entire collection to the DDS bus. This avoids two problems related to using an unbounded sequence type in a DDS message: 1) resource consumption growing as the collection is appended to or updated, and 2) DDS implementation-specific limitations on unbounded sequences. There are two implementations of a Large Collection: the Large Set (unordered) and the Large List (ordered).

In both Large Collection implementations, there are two important abstractions: the collection metadata and collection element type. Because Large Collections are specific to the UMAA PSM, the type definitions for the collection metadata and collection element are not part of MDE, and the IDL definitions of these types are generated separately. A particular UMAA message that has a Large Collection attribute will reference the metadata type (LargeSetMetadata or LargeListMetadata). The collection element type is defined under the same namespace as the message that uses it, and follows the naming pattern <parent message name><attribute name><collection type>Element. Each element of the collection is published as a separate message on the DDS bus, and can be tracked back to their related collection using the setID or listID. Users can also trace an element in a set to the source attribute (a NumericGUID) of the Service Provider that generated the report with this set using the collection metadata.

3.8.1 Necessary QoS

To achieve the Large Collection consistency in the update process described below, ordering of samples on the collection element type topic is necessary. Therefore, publishers and subscribers to the collection element type topic must use the `PRESENTATION` QoS policy with an `access_scope` of `DDS_TOPIC_PRESENTATION_QOS` and `ordered_access`.

3.8.2 Updating Large Collections

When elements of the collection are updated, the metadata must be updated as well to signal a change in the set. The `updateElementID` is updated to match the elementID of the element whose reception signals the end of the atomic update of the collection. Because of the requirement of an ordered topic described above, this will be the element that is updated last chronologically. The metadata `updateElementTimestamp` must be updated to the timestamp of the same element that signals the end of the update.

The set can be updated as a batch (multiple elements in a single "update cycle," as determined by the provider). This allows for a coarse synchronization: data elements that do not match the metadata `updateElementID` and `updateElementTimestamp` can be assumed to be part of an in-progress update cycle. Consumers can choose to immediately act on those data individually or wait until the matching element is received to signal that the complete update cycle has finished and consider the set as a whole. Note that the coarseness of synchronization is service-dependent: in some cases an intermediate view of a collection update may be logically incorrect to act upon.

3.8.3 Specifying an Empty Large Collection

A particular Large Collection can be empty during initial creation. This is indicated by publishing metadata with a `size` of zero and an `updateElementID` set to the Nil UUID. As specified in section 4.1.7 of the referenced document "A Universally Unique IDentifier (UUID) URN Namespace", this is a "special form of UUID that is specified to have all 128 bits set to zero".

3.8.4 Large Set Types

The following details the LargeSetMetadata structure:

Table 4: LargeSetMetadata Structure Definition

Attribute Name	Attribute Type	Attribute Description
setID	NumericGUID	Identifies the Large Set instance this metadata relates to.
updateElementID	NumericGUID	This field references the element ID of the set element whose reception signals the end of an atomic update to this set. This elementID must be used in conjunction with the updateElementTimestamp below to fully identify when the atomic update has completed and the set is stable.
updateElementTimestamp†	DateTime	This field identifies the elementTimestamp of the element, referenced above by updateElementID, that signals the end of an atomic update to this set. This field will be empty in the event that the element update results from a DDS dispose.
size	LargeCollectionSize	Indicates the number of elements associated with this set after the atomic update is complete.

An example element type is shown below, where a `FooReportType` message has a Large Set attribute called "items" whose type is `BarType`

Table 5: Example FooReportTypeItemsSetElement Structure Definition

Attribute Name	Attribute Type	Attribute Description
element	BarType	The value of the set element.
setID	NumericGUID	Identifies the Large Set instance this element relates to.
elementID*	NumericGUID	Uniquely identifies this element within the set and across all large collection elements that currently exist on the DDS bus.
elementTimestamp	DateTime	The timestamp of this element.

3.8.5 Large List Types

The following details the LargeListMetadata structure:

Table 6: LargeListMetadata Structure Definition

Attribute Name	Attribute Type	Attribute Description
listID	NumericGUID	Identifies the Large List instance this metadata relates to.
updateElementID	NumericGUID	This field references the element ID of the list element whose reception signals the end of an atomic update to this list. This elementID must be used in conjunction with the updateElementTimestamp below to fully identify when the atomic update has completed and the list is stable.
updateElementTimestamp†	DateTime	This field identifies the elementTimestamp of the element, referenced above by updateElementID, that signals the end of an atomic update to this list. This field will be empty in the event that the element update results from a DDS dispose.
startingElementID	NumericGUID	This field identifies the list element, tying to its elementID, that is sequentially first in the list. This is provided for convenience when iterating through the linked list using the nextElementID field.
size	LargeCollectionSize	Indicates the number of elements associated with this set after the atomic update is complete.

An example element type is shown below, where a `FooReportType` message has a Large List attribute called "items" whose type is `BarType`

Table 7: Example FooReportTypeItemsListElement Structure Definition

Attribute Name	Attribute Type	Attribute Description
element	BarType	The value of the list element.
listID	NumericGUID	Identifies the Large List instance this element relates to.
elementID*	NumericGUID	Uniquely identifies this element within the list and across all large collection elements that currently exist on the DDS bus.
elementTimestamp	DateTime	The timestamp of this element.
nextElementID†	NumericGUID	This field references to the elementID of the element that logically follows this element in the linked list. This is empty if this element is sequentially last.

4 Flow Control

4.1 Command / Response

This section defines the flow of control for command/response over the DDS bus. A command/response controls a specific service. While the exact names and processes will depend on the specific service and command being executed, all command/responses in UMAA follow a similar pattern. A notional "Function" command `FunctionCommand` is used in the following examples. As will be described in subsequent paragraphs, DDS publish/subscribe methods are used in implementations to issue commands and responses.

To direct a `FunctionCommand` at a specific Service Provider, UMAA includes a `destination` GUID in all commands. A Service Provider is required to respond to all `FunctionCommand`s where the `destination` is the same as the Service Provider's ID. The Service Consumer will also create a `sessionID` for the command when commanded. The `sessionID` is used to track the command execution as a key into other command-related messages. The `sessionID` must be unique across all `FunctionCommand` instances that are active (i.e. currently on the DDS bus), otherwise the Service Provider will consider the `FunctionCommand` to be a command update (see Section 4.1.4.2). Once a `FunctionCommand` is removed from the DDS bus as part of the Command Cleanup process (see Section 4.1.5), its `sessionID` may be reused for future commands without triggering a command update; therefore it is not necessary for a Service Provider to maintain a complete history of `sessionID`s.

Service Provider and Service Consumer terminology in the following sections is adopted from the OMG Service-oriented architecture Modeling Language (SoaML).

To initialize, a Service Provider (controllable resource) subscribes to the `FunctionCommand` DDS topic. At startup or right before issuing a command, the Service Consumer (controlling resource) subscribes to the `FunctionCommandStatus` DDS topic. Optionally, the Service Consumer may also subscribe to the `FunctionCommandAckReport` to monitor which command is currently being executed, and the `FunctionExecutionStatusReport` (if defined for the Function service) that provides reporting on function-specific data status.

Both Service Providers and Service Consumers are required to recover or clean up any previous persisted commands on the bus during initialization.

To execute a command, the Service Consumer publishes a `FunctionCommandType` to the DDS bus. The Service Provider will be notified and will begin processing the request. During each phase of processing, the Service Provider will provide updates to the Service Consumer via published updates to a related `FunctionCommandStatus` topic. Command responses are correlated to their originating command via the `sessionID`. If a command with a duplicate `sessionID` is received, the Service Provider will regard this as a command update, and follow the flow control detailed in Section 4.1.4.2. Command status updates are provided in the command responses via the `commandStatus` field with additional details included in the `commandStatusReason` field. The Service Provider will also publish the current executing command to the `FunctionCommandAckReport` topic. When defined for the Function service, the Service Provider must also publish the `FunctionExecutionStatusReport` topic and update it as appropriate throughout the execution of the command.

The required state transitions for the `commandStatus` field are shown in Figure 4. Commands may complete normally, or they may terminate early due to failure (Section 4.1.4.4) or cancellation (Section 4.1.4.5). The state machine for a command can also be reset to `ISSUED` via a command update (Section 4.1.4.2). If there is not a self-transition indicated in the diagram, you cannot republish that state in a message. Every command must transition through the states as defined. For example, it is a violation to transition from `ISSUED` to `EXECUTING` without transitioning through `COMMANDED`. Even in the case where there is no logic executing between the `ISSUED` and `EXECUTING` states, the Service Provider is required to transition through `COMMANDED`. This ensures consistent behavior across different Service Providers, including those that do require the `COMMANDED` state.

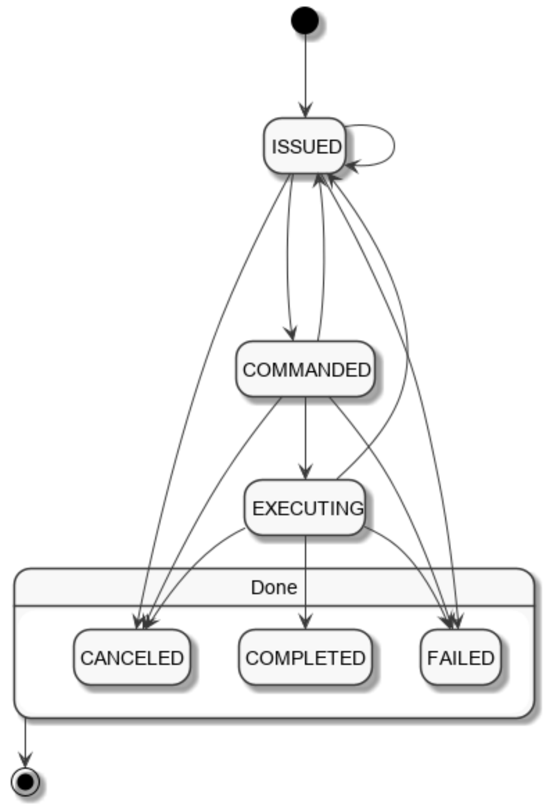

Figure 4: State transitions of the `commandStatus` as commands are processed.

As described above, each time a command transitions to a new state, a `FunctionCommandStatus` message is published containing the updated `commandStatus` and a `commandStatusReason` that indicates why the state transition happened. The table below shows all valid `commandStatusReason` values for each `commandStatus` transition.

	Ending State					
Starting State	ISSUED	COMMANDED	EXECUTING	COMPLETED	FAILED	CANCELED
Initial State	SUCCEEDED	—	—	—	—	—
ISSUED	UPDATED	SUCCEEDED	—	—	VALIDATION_FAILED RESOURCE_FAILED INTERRUPTED TIMEOUT SERVICE_FAILED	CANCELED
COMMANDED	UPDATED	—	SUCCEEDED	—	RESOURCE_REJECTED INTERRUPTED TIMEOUT SERVICE_FAILED	CANCELED
EXECUTING	UPDATED	—	—	SUCCEEDED	OBJECTIVE_FAILED RESOURCE_FAILED INTERRUPTED TIMEOUT SERVICE_FAILED	CANCELED
COMPLETED	—	—	—	—	—	—
FAILED	—	—	—	—	—	—
CANCELED	—	—	—	—	—	—

Figure 5: Valid `commandStatusReason` values for each `commandStatus` state transition. Entries marked with a (—) indicate that the state transition is invalid.

In the following sections, the sequence diagrams demonstrate different exchanges between a Service Consumer and Service

Provider. Within the diagrams, the dashed arrows represent implementation-specific communications that are outside of UMAA's scope. These sequence diagrams are just an example of one possible implementation. Other implementations may have different communication patterns between the Service Provider and the Resource or be implemented completely within the Service Provider process itself (no dependency on an external Resource). Likewise, the interactions between the User and Service Consumer may follow similar or different patterns. However, the UMAA-defined exchanges with the DDS bus between the Service Consumer and Service Provider must happen in the order shown within the sequence diagrams.

4.1.1 High-Level Flow

The high-level flow of a command sequence is shown in Figure 6 and can be described as follows:

1. The Command Startup Sequence is performed.
2. For each command to be executed:
   (a) The Command Start Sequence is performed.
   (b) The command is executed (sequence depends on the execution path, i.e., success, failure, or cancel).
   (c) The Command Cleanup Sequence is performed.
3. The Command Shutdown Sequence is performed.

The `ref` blocks will be defined in later sequence diagrams. Note that the duration of the system execution for any particular `FunctionCommandType` is defined by the combination of the Service Provider(s) and Service Consumer(s) in the system and may not be identical to the overall system execution duration. For example, providers may only be available to execute certain commands during specific mission phases or when certain hardware is in specific configurations. This Command Startup Sequence is not required to happen during a system startup phase. The only requirement is that it must be completed by at least one Service Provider and one Service Consumer before any `FunctionCommandType` commands can be fully executed. Likewise, the Command Shutdown sequence may occur at any time the `FunctionCommandType` will no longer be supported. There is no requirement stating that the Command Shutdown Sequence only be performed during a system shutdown phase.

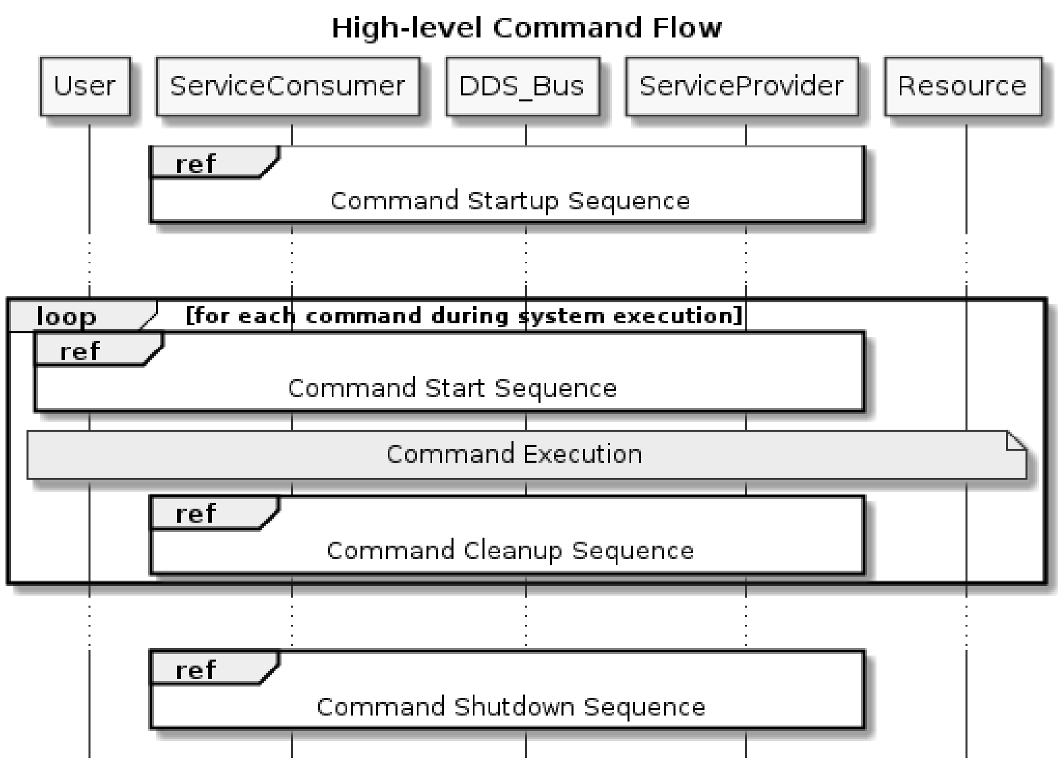

Figure 6: Sequence Diagram for the High-Level Description of a Command Execution.

4.1.2 Command Startup Sequence

As part of initialization both the Service Provider and Service Consumer are required to perform a startup sequence. This startup prepares the Service Provider to execute commands and the Service Consumer to request commands and monitor the progress of those requested commands.

The Service Provider and Service Consumer can initialize in any order. Commands will not be completely executed until both have completed their initialization. The sequence diagram is shown in Figure 7.

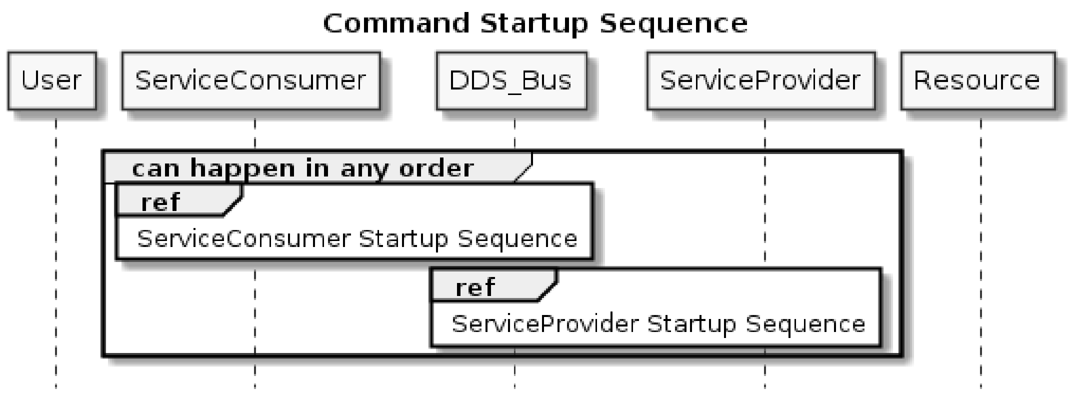

Figure 7: Sequence Diagram for Command Startup.

4.1.2.1 Service Provider Startup Sequence During startup, the Service Provider is required to register as a publisher to the `FunctionCommandStatus`, `FunctionCommandAckReport`, and (if defined for the Function service) the `FunctionExecutionStatusReport` topics.

The Service Provider is also required to subscribe to the `FunctionCommand` topic to be notified when new commands are published.

Finally, the Service Provider is required to handle any existing `FunctionCommandType` commands persisted on the DDS bus with the Service Provider's ID. For each command, if the Service Provider can and wishes to recover, it can continue to execute the command. To obtain the last published state of the command, the Service Provider must subscribe to the `FunctionCommandStatusType`. The Service Provider will continue following the normal status update sequence, picking up from the last status on the bus. If the Service Provider cannot or chooses not to continue processing the command, it must fail the command by publishing a `FunctionCommandStatus` with a `commandStatus` of `FAILED` and a `reason` of `SERVICE_FAILED`.

The Service Provider Startup sequence is shown in Figure 8.

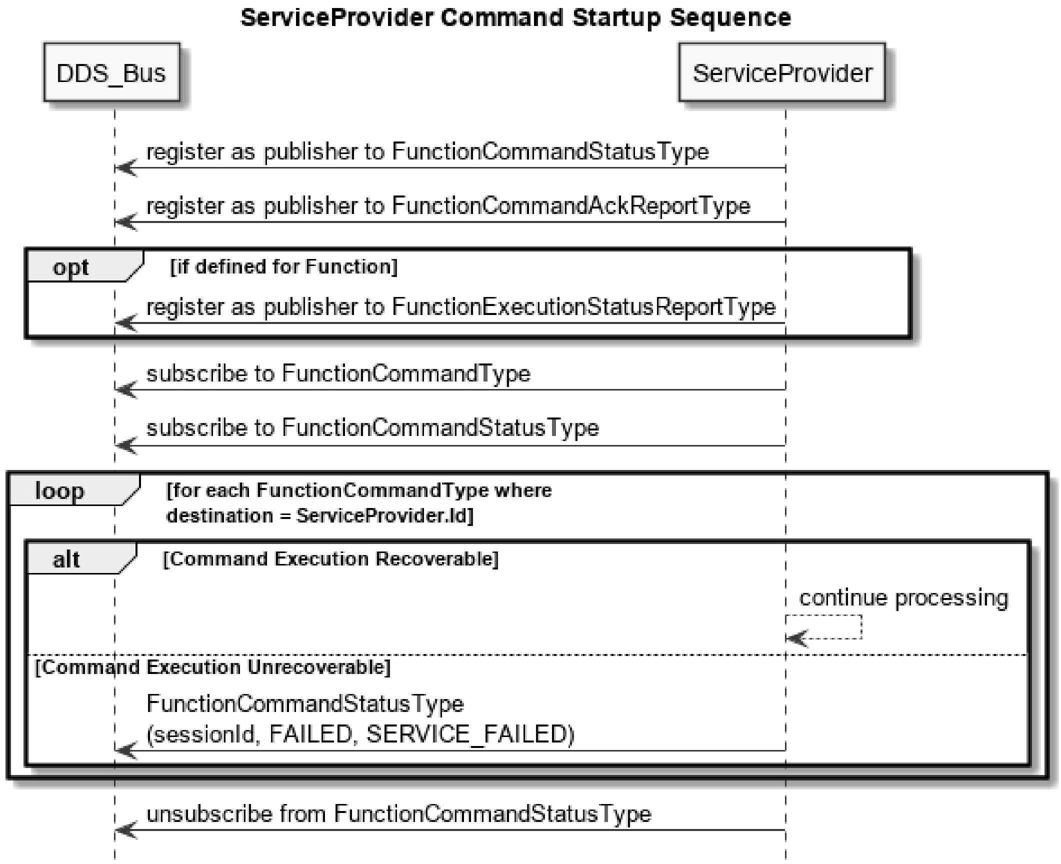

Figure 8: Sequence Diagram for Command Startup for Service Providers.

4.1.2.2 Service Consumer Startup Sequence

During startup, the Service Consumer is required to register as a publisher of the `FunctionCommandType`.

The Service Consumer is also required to subscribe to the `FunctionCommandStatusType` to monitor the execution of any published commands. The Service Consumer can optionally register for the `FunctionCommandAckReportType` and, if defined for the Function service, the `FunctionExecutionStatusReportType` if it desires to track additional status of the execution of commands.

Finally, the Service Consumer is required to handle any existing `FunctionCommandType` commands persisted on the DDS bus with this Service Consumer's ID. To find existing `FunctionCommandType`s on the bus, it must first subscribe to the topic. If the Service Consumer can and wishes to recover, it can continue to monitor the execution of the command. If the Service Consumer cannot or chooses not to continue the execution of the command, it must cancel the command via the normal command cancel method.

The Service Consumer Startup sequence is shown in Figure 9.

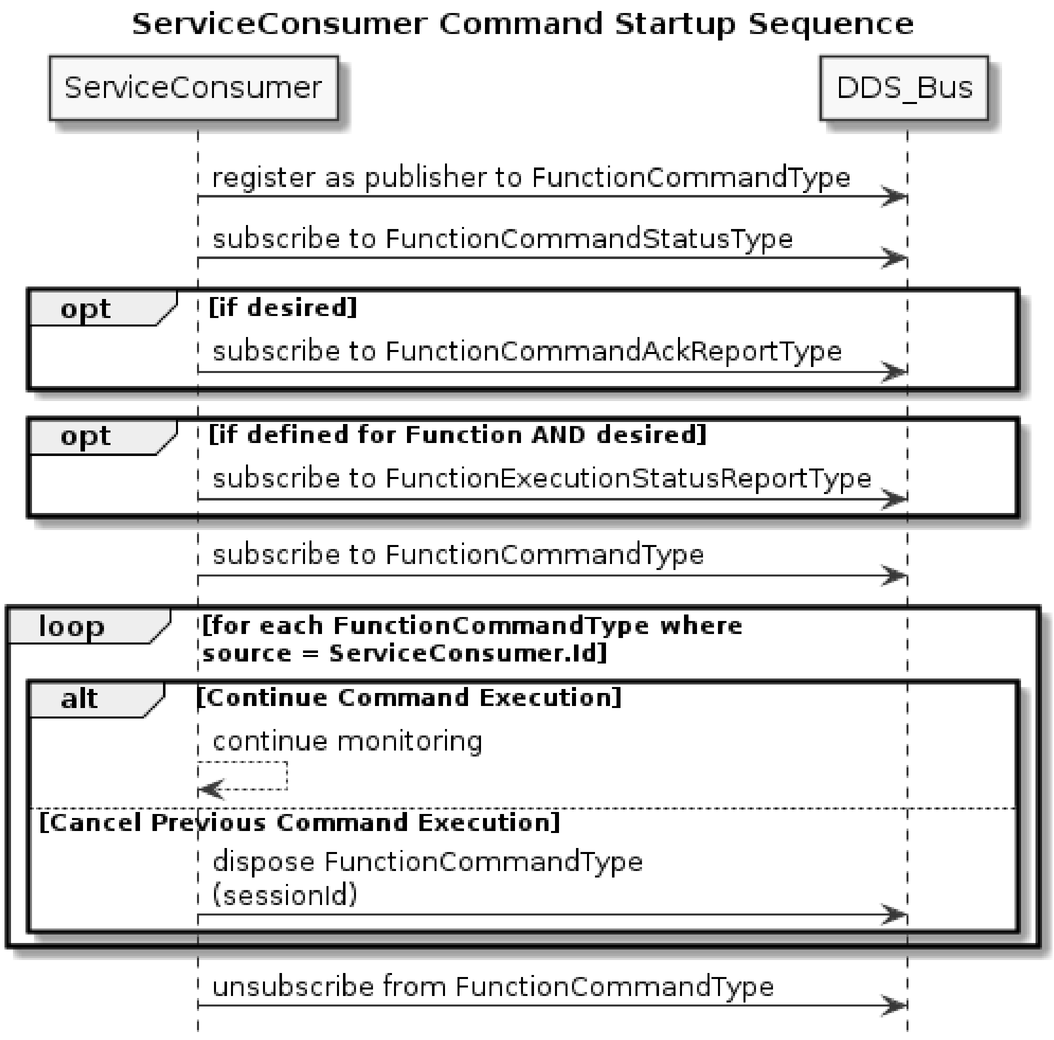

Figure 9: Sequence Diagram for Command Startup for Service Consumers.

4.1.3 Command Execution Sequences

Once both the Service Provider and Service Consumer have performed the startup sequence, the system is ready to begin issuing and executing commands.

4.1.4 Command Start Sequence

The initial start sequence to execute a single new command follows this pattern:

1. The User of the Service Consumer issues a request for a command to be executed.
2. The Service Consumer publishes the `FunctionCommandType` with a unique session ID, the source ID of the Service Consumer, and the destination ID of the desired Service Provider.
3. The Service Provider, upon notification of the new `FunctionCommandType`, publishes a new `FunctionCommandStatusType` with (1) the same session ID as the new `FunctionCommandType`, (2) the status of `ISSUED` and (3) the reason of `SUCCEEDED` to notify the Service Consumer it has received the new command.

The Command Start Sequence for a new command is shown in Figure 10. This pattern will be repeated each time a new command is requested. Note that the Command Start Sequence differs if the `FunctionCommandType` has a `sessionID` that matches another `FunctionCommandType` that currently exists on the DDS bus. This is considered a command update and detailed in Section 4.1.4.2.

After the Command Start Sequence, the sequence can take different paths depending on the actual execution of the command, detailed from Section 4.1.4.1 to Section 4.1.4.5, but they do not enumerate all of the possible execution paths. Other paths (e.g., an objective failing) will follow a similar pattern to other failures; all are required to follow the state diagram shown in Figure 4 and eventually end with the Command Cleanup Sequence (shown in Figure 17).

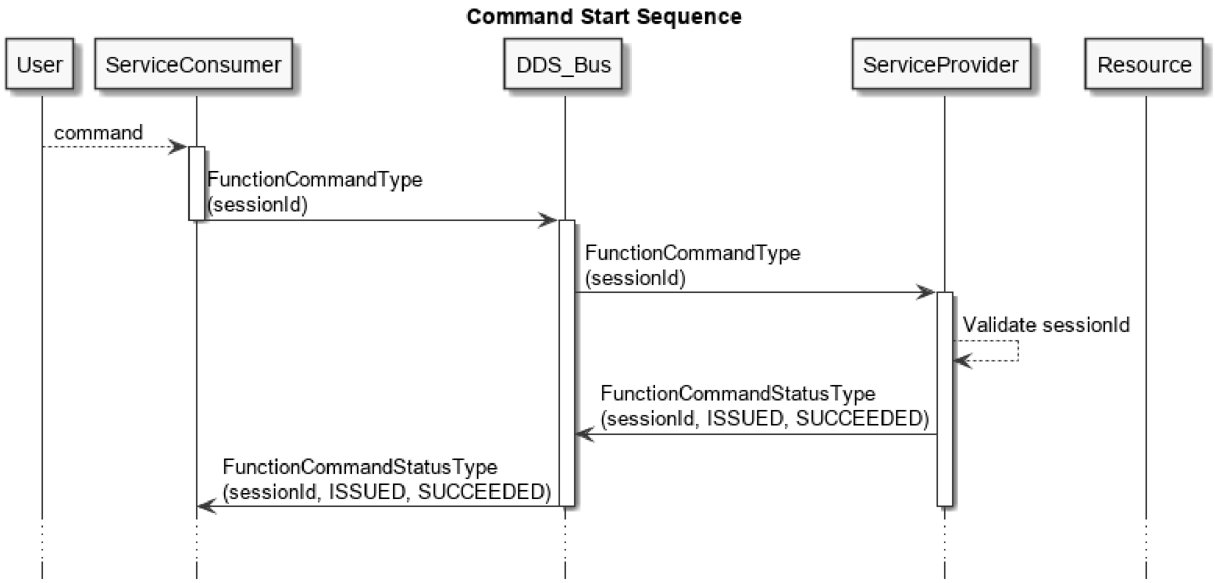

Figure 10: Sequence Diagram for the Start of a Command Execution.

4.1.4.1 Command Execution

Once a Service Provider starts to process a command, the Command Execution sequence is:

1. The Service Provider publishes a `FunctionCommandAckReportType` with matching session ID and parameters as the `FunctionCommandType` it is starting to process.
2. The Service Provider performs any validation and negotiation with backing resources as necessary. Once the command is ready to be executed, the Service Provider publishes a `FunctionCommandStatusType` with a status `COMMANDED` and reason `SUCCEEDED` to notify the Service Consumer that the command has been validated and commanded to start execution.
3. Once the command has begun executing, the Service Provider publishes a `FunctionCommandStatusType` with a status `EXECUTING` and reason `SUCCEEDED` to notify the Service Consumer that the command has been validated and commanded to start.
4. If the Function has a defined `FunctionExecutionStatusReportType`, the Service Provider must publish a new instance with matching session ID as the associated `FunctionCommandType`. The `FunctionExecutionStatusReportType` must be updated by the Service Provider throughout the execution as dictated by the definitions of the command-specific attributes in the execution status report.

The command execution sequence is shown in Figure 11. This sequence holds until the command completes execution.

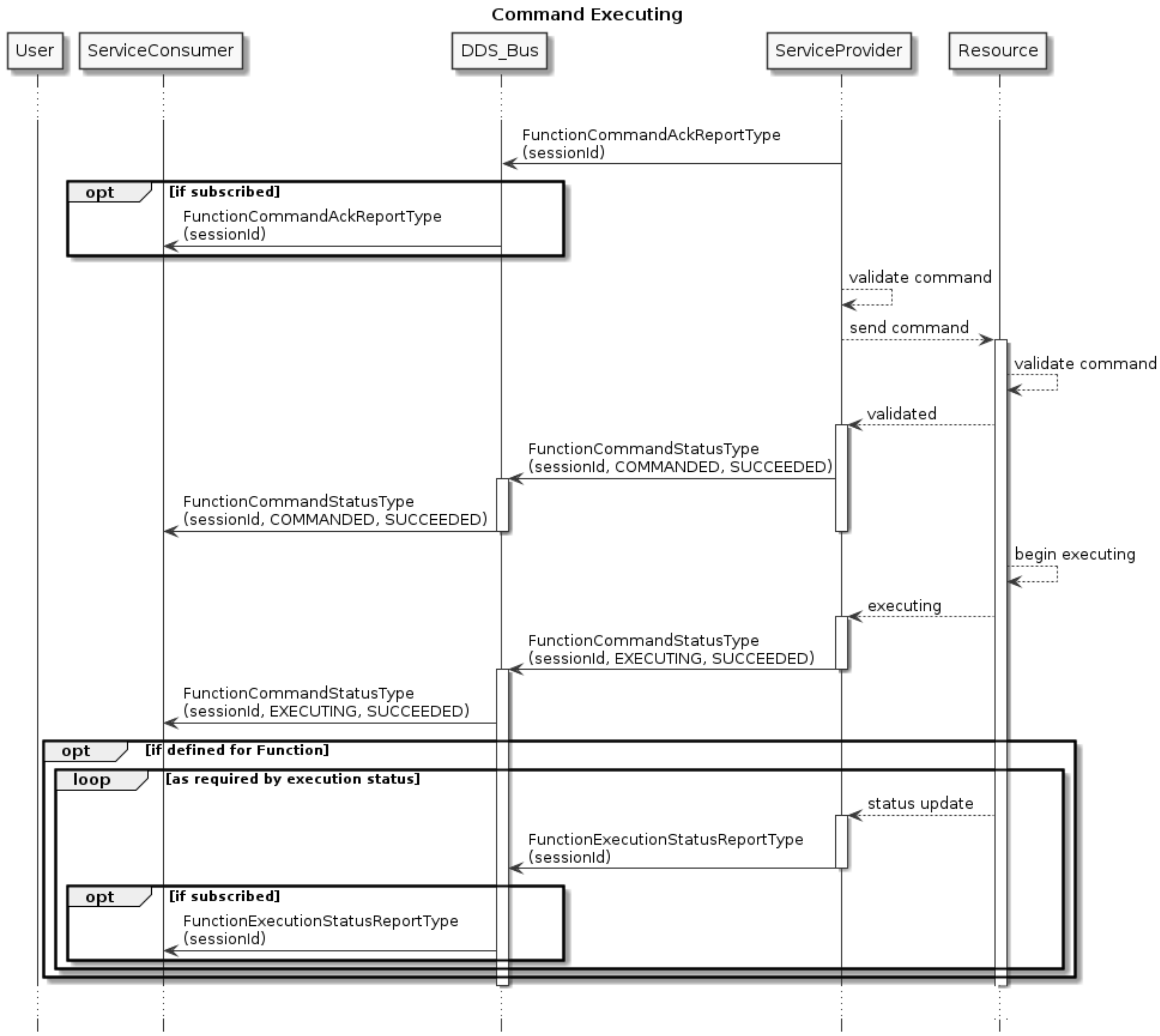

Figure 11: Beginning Sequence Diagram for a Command Execution.

The normal successful conclusion of a command being executed in some cases is initiated by the Service Consumer (an endless GlobalVector command concluded by canceling it) and in other cases is initiated by the Service Provider (a GlobalWaypoint commanded concluded by reaching the last waypoint). Unless otherwise explicitly stated, it is assumed the Service Provider will be able to identify the successful conclusion of a command. In the cases where commands are defined to be indeterminate the Service Consumer must cancel the command when the Service Consumer no longer desires the command to be executed.

4.1.4.2 Updating a Command An updated command is defined as a command with a source ID and session ID identical to the current command being processed by the Service Provider, but whose timestamp is newer than the current command. Only a command that is in a non-terminal state may be updated - otherwise, the Service Consumer must follow the normal command cleanup process and issue a new command with an updated unique session ID. When the Service Provider receives an updated command, it is required to take one of two possible actions:

1. If the current command is in a non-terminal state (`ISSUED`, `COMMANDED`, or `EXECUTING`), then the Service Provider publishes a `FunctionCommandStatusType` with a status `ISSUED` and reason `UPDATED`. The state machine then restarts and proceeds through the normal command flow detailed in 4.1.4. The Service Provider must consider the updated command as an entirely new command, resetting any internal state related to the command (e.g. a timer that keeps track of command timeout).
2. If the current command is in a terminal state (`COMPLETED`, `CANCELED`, or `FAILED`), then the updated command cannot be processed, and the Service Provider must publish a `FunctionCommandStatusType` with a status `FAILED` and follow the normal command cleanup process.

The flow control for command update is detailed below:

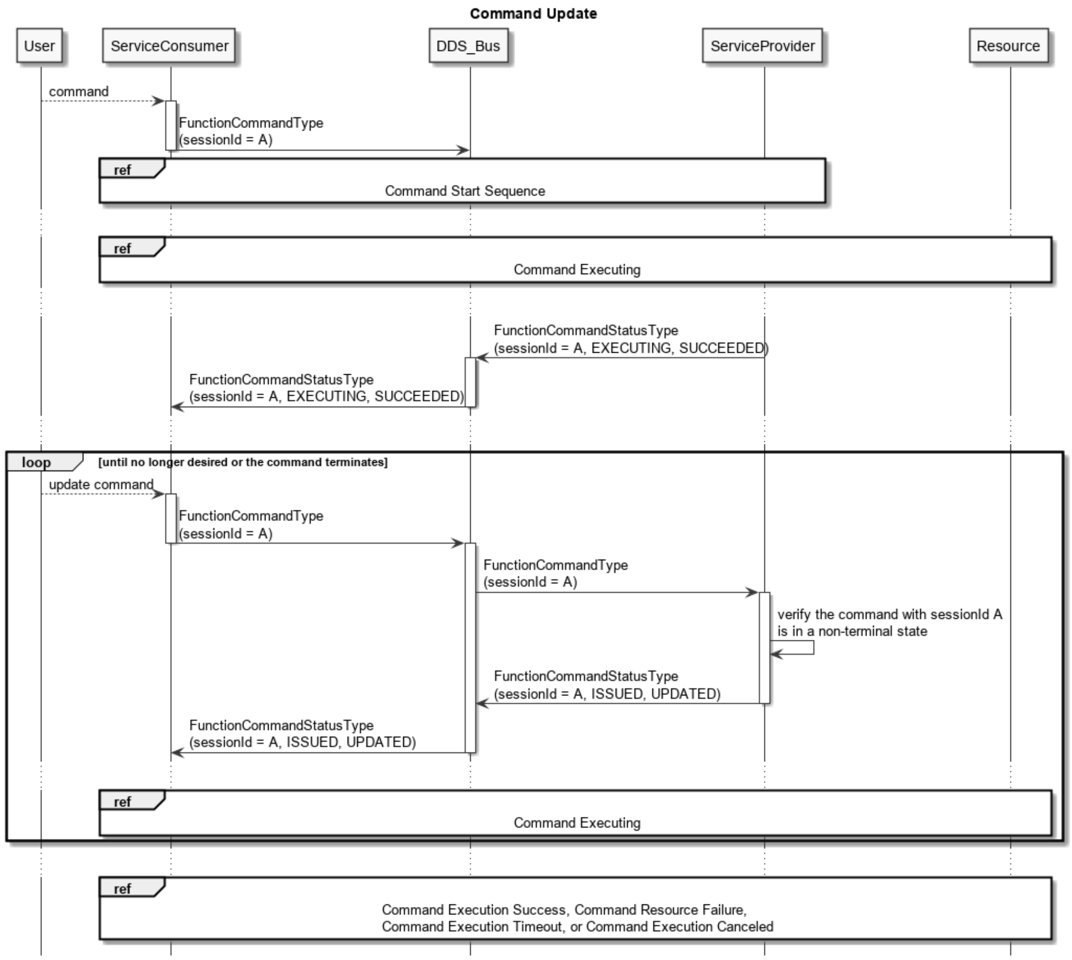

Figure 12: Sequence Diagram for Command Update.

4.1.4.3 Command Execution Success When the Service Provider determines a command has successfully completed, it must update the associated `FunctionCommandStatusType` with as status of `COMPLETED` and reason of `SUCCEEDED`. This signals to the Service Consumer that the command has completed successfully.

The Command Execution Success sequence is shown in Figure 13.

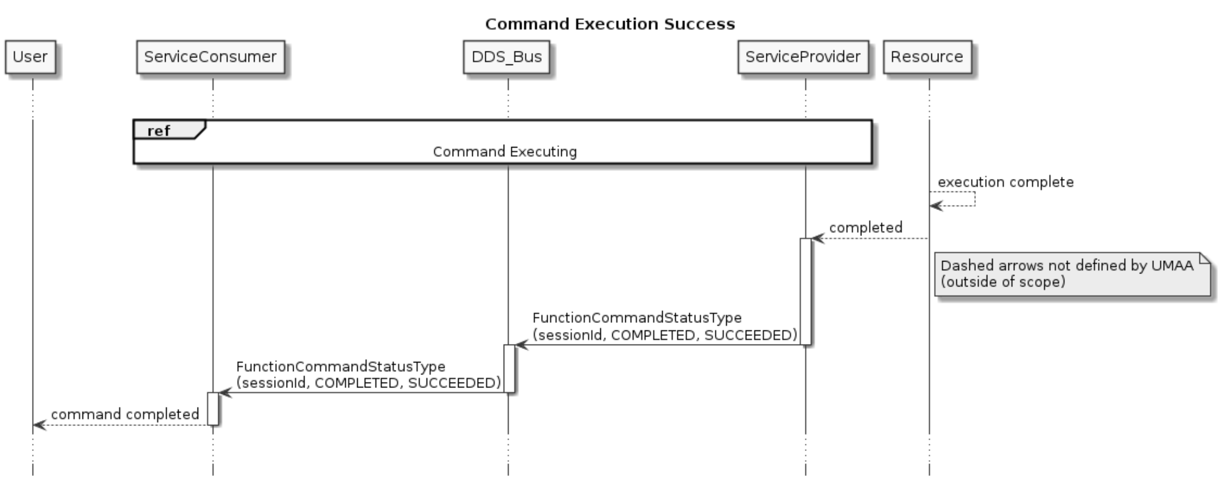

Figure 13: Sequence Diagram for a Command That Completes Successfully.

4.1.4.4 Command Execution Failure The command may fail to complete for any number of reasons including software errors, hardware failures, or unfavorable environmental conditions. The Service Provider may also reject a command for a number of reasons including inability to perform the task, malformed or out of range requests, or a command being interrupted by a higher priority process. In all cases, the Service Provider must publish a `FunctionCommandStatusType` with an identical `sessionID` as the originating `FunctionCommandType` with a status of `FAILED` and the reason that reflects the cause of the failure (`VALIDATION_FAILED, SERVICE_FAILED, OBJECTIVE_FAILED,` etc).

Figure 14 and Figure 15 provide examples where a command has failed.

In the first example, the backing Resource failed and the Service Provider is unable to communicate with it. In this case, the Service Provider will report a `FunctionCommandStatusType` with a status of `FAILED` and a reason of `RESOURCE_FAILED`. This is shown in Figure 14.

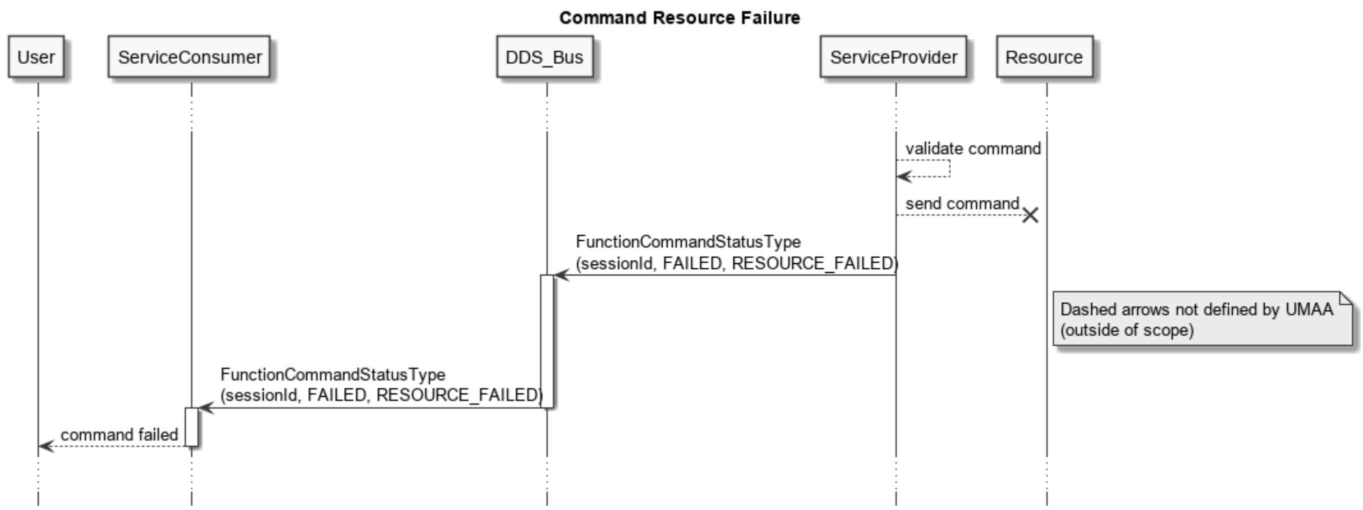

Figure 14: Sequence Diagram for a Command That Fails due to Resource Failure.

In the second example, the Resource takes too long to respond, so the Service Provider cancels the request and reports a `FunctionCommandStatusType` with a status of `FAILED` and a reason of `TIMEOUT`. This is shown in Figure 15.

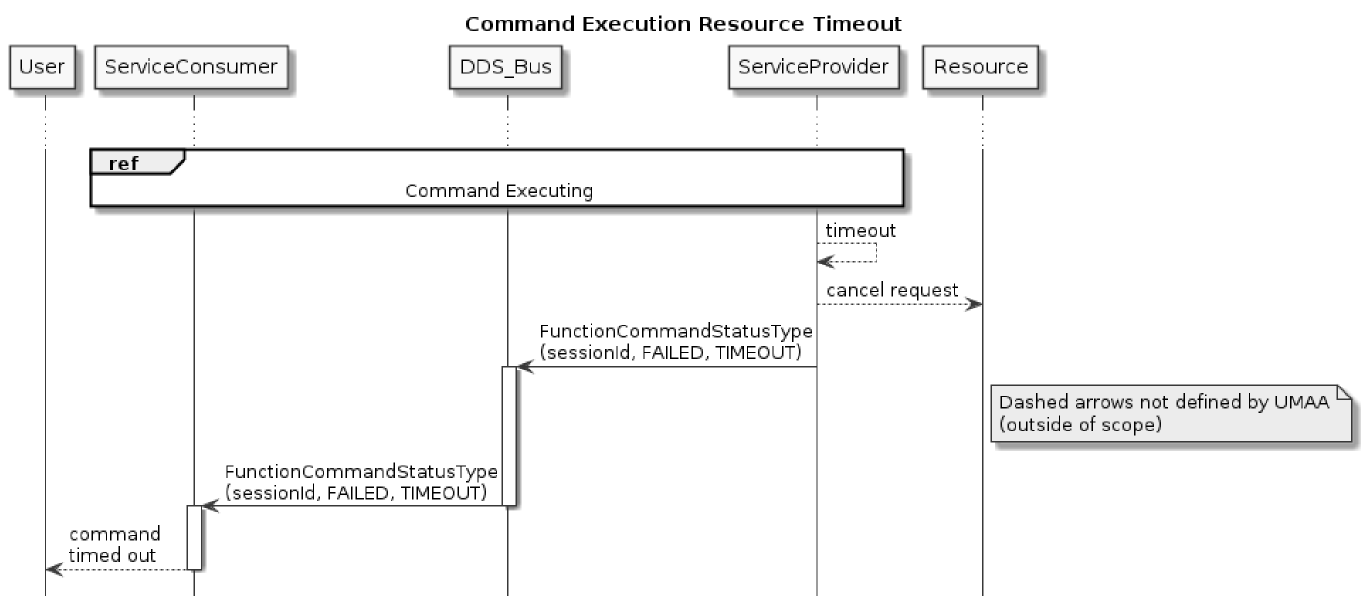

Figure 15: Sequence Diagram for a Command That Times Out Before Completing.

Other failure conditions will follow a similar pattern: when the failure is recognized, the Service Provider will publish a

FunctionCommandStatusType with a status of FAILED and a reason that reflect the cause of the failure.

4.1.4.5 Command Canceled The Service Consumer may decide to cancel the command before processing is finished. To signal a desire to cancel a command, the Service Consumer disposes of the existing FunctionCommandType from the DDS bus before the execution is complete. When notified of the command disposal, and if the Service Provider is able to cancel the command, it should respond to the Service Consumer with a FunctionCommandStatusType with both the status and reason as CANCELED. At this point, the DDS bus should dispose of the FunctionCommandStatusType, the FunctionCommandAckReportType and, (if defined for the Function service) the FunctionExecutionStatusReportType. This is shown in Figure 16. If the command cannot be canceled, then the Service Provider can continue to update the command status until the execution is completed. Reporting will include FunctionCommandStatusType with a status of COMPLETED and a reason of SUCCEEDED. Then, the DDS bus should dispose of the FunctionCommandStatusType, the FunctionCommandAckReportType, and (if defined for the Function service) the FunctionExecutionStatusReportType.

There is no new, unique, or specific status message response to a cancel command from the Service Provider. The cancel command status can be inferred through the corresponding FunctionCommandStatusType status and reason updates.

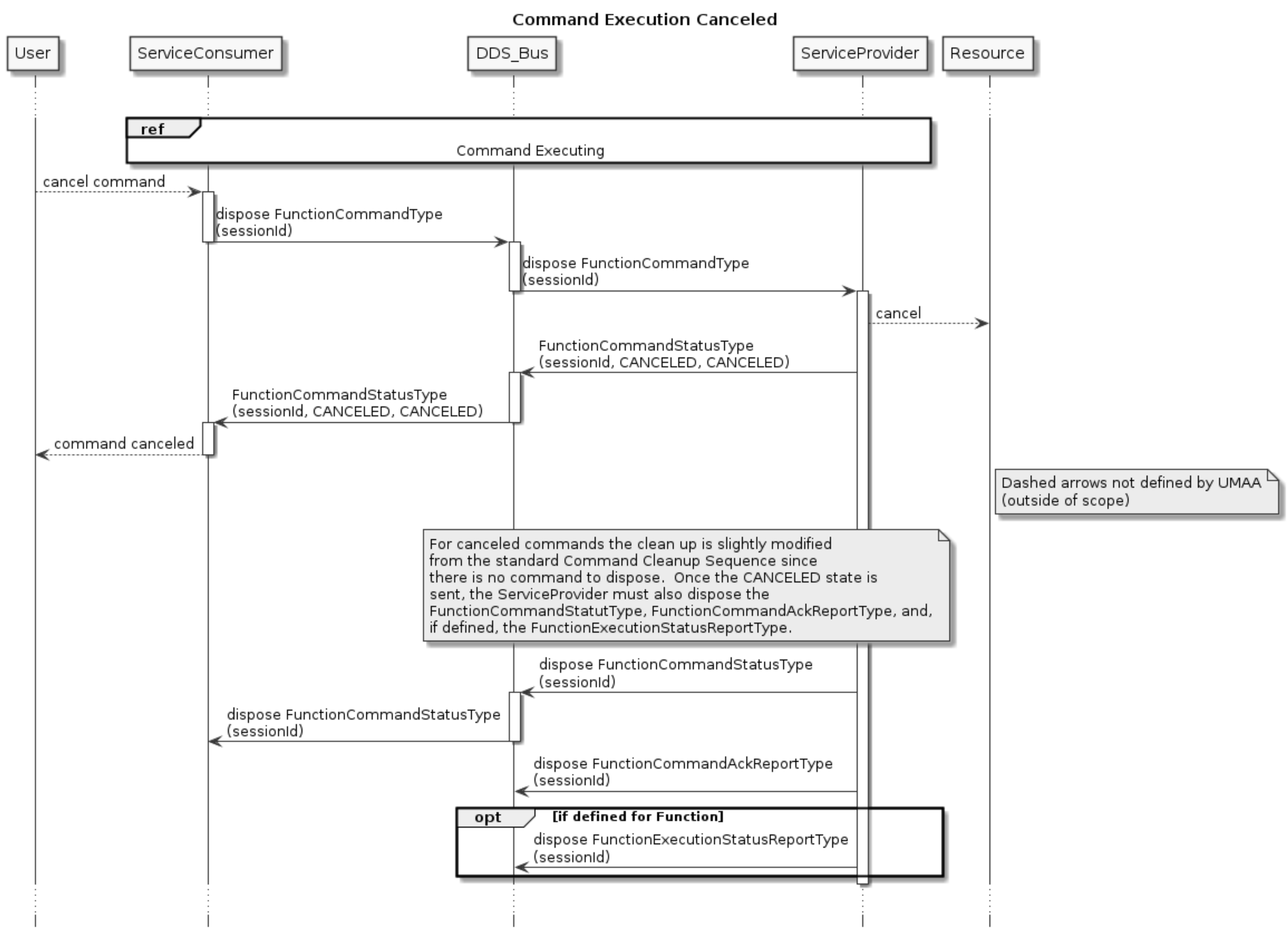

Figure 16: Sequence Diagram for a Command That is Canceled by the Service Consumer Before the Service Provider can Complete It.

4.1.5 Command Cleanup

The Service Consumer and Service Provider are responsible for disposing of corresponding data that is published to the DDS bus when the command is no longer active. With the exception of a canceled command, the signal that a FunctionCommandType can be disposed is when the FunctionCommandStatusType reports a terminal state (COMPLETED or FAILED)[3]. In turn, the

[3]While CANCELED is also a terminal state, the CANCELED command cleanup is handled specially as part of the cancelling sequence and, as such, does not need to be handled here.

signal that a `FunctionCommandStatusType`, `FunctionCommandAckReportType`, and (if defined for the Function service) the `FunctionExecutionStatusReportType` can be disposed is when the corresponding `FunctionCommandType` has been disposed. This is shown in Figure 17.

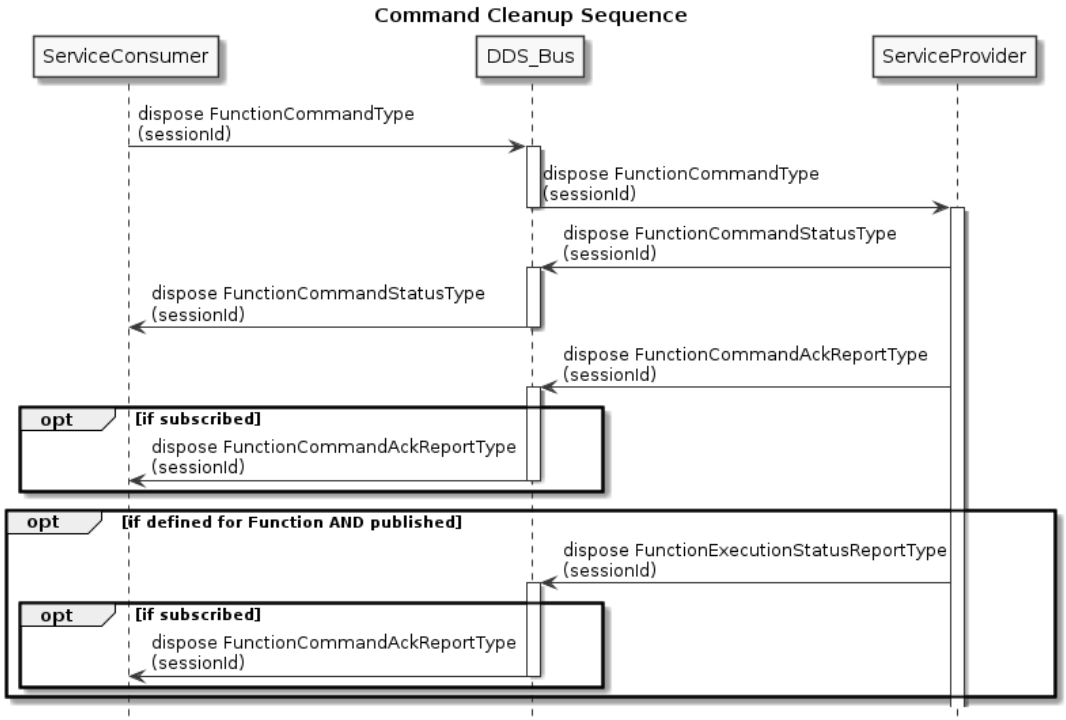

Figure 17: Sequence Diagram Showing Cleanup of the Bus When a Command Has Been Completed and the Service Consumer No Longer Wishes to Maintain the Commanded State.

4.1.6 Command Shutdown Sequence

As part of shutdown, both the Service Provider and Service Consumer are required to perform a shutdown sequence. This shutdown cleans up resources on the DDS bus and informs the system that the Service Provider and Service Consumer are no longer available.

The Service Provider and Service Consumer can shut down in any order. The sequence diagram is shown in Figure 18.

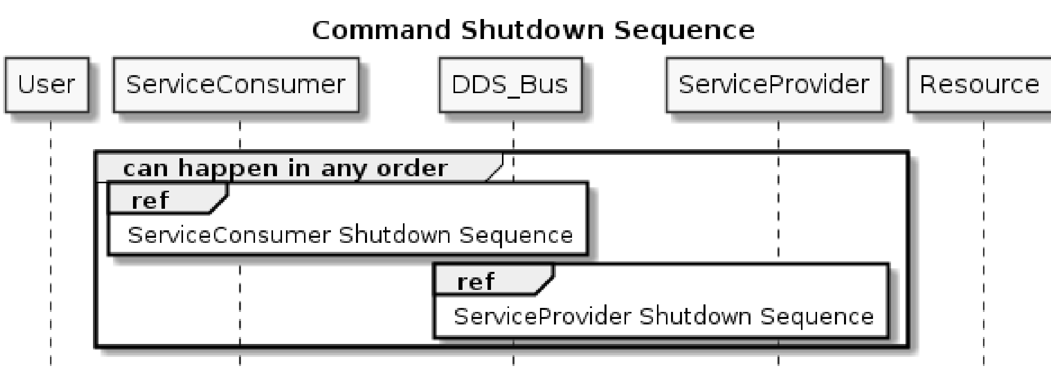

Figure 18: Sequence Diagram for Command Shutdown.

4.1.6.1 Service Provider Shutdown Sequence During shutdown, the Service Provider is required to fail any incomplete requests and then unregisters as a publisher of the `FunctionCommandStatusType`, `FunctionCommandAckReportType`, and (if defined for the Function service) the `FunctionExecutionStatusReportType`.

The Service Provider is also required to unsubscribe from the `FunctionCommandType`.

The Service Provider Shutdown sequence is shown in Figure 19.

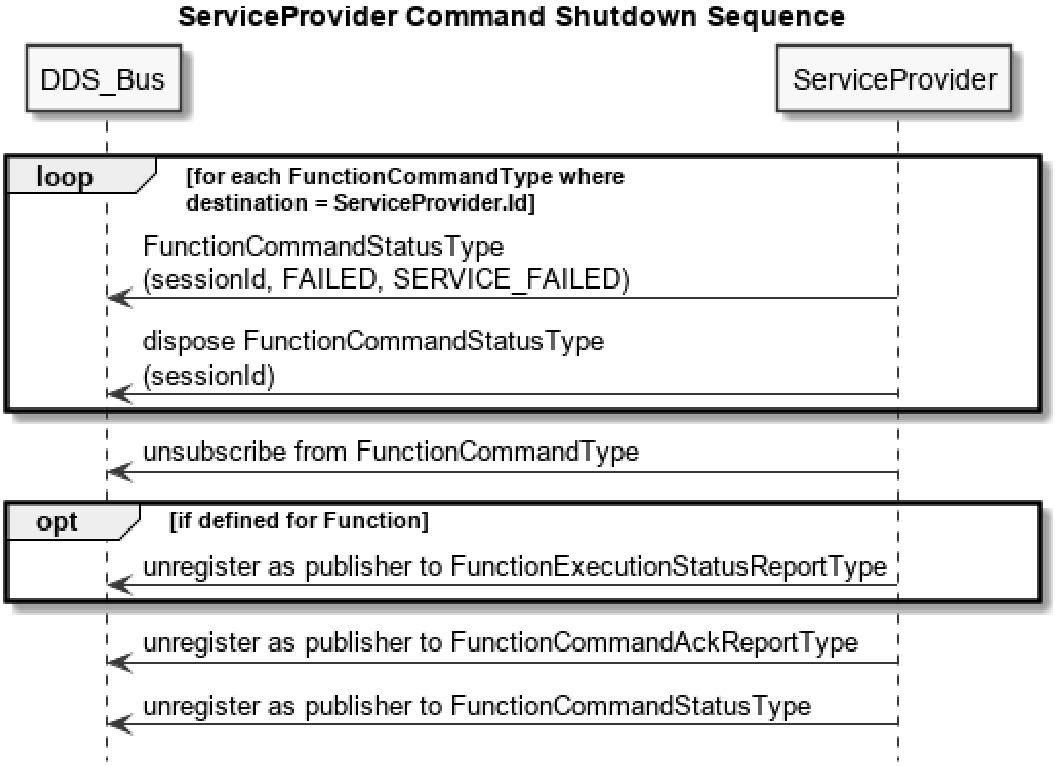

Figure 19: Sequence Diagram for Command Shutdown for Service Providers.

4.1.6.2 Service Consumer Shutdown Sequence During shutdown, the Service Consumer is required to cancel any incomplete requests and then unregister as a publisher of the `FunctionCommandType`.

The Service Consumer is also required to unsubscribe from the `FunctionCommandStatusType`, the `FunctionCommandAckReportType` if subscribed, and the `FunctionExecutionStatusReportType` if defined for the Function service and subscribed.

The Service Consumer Shutdown sequence is shown in Figure 20.

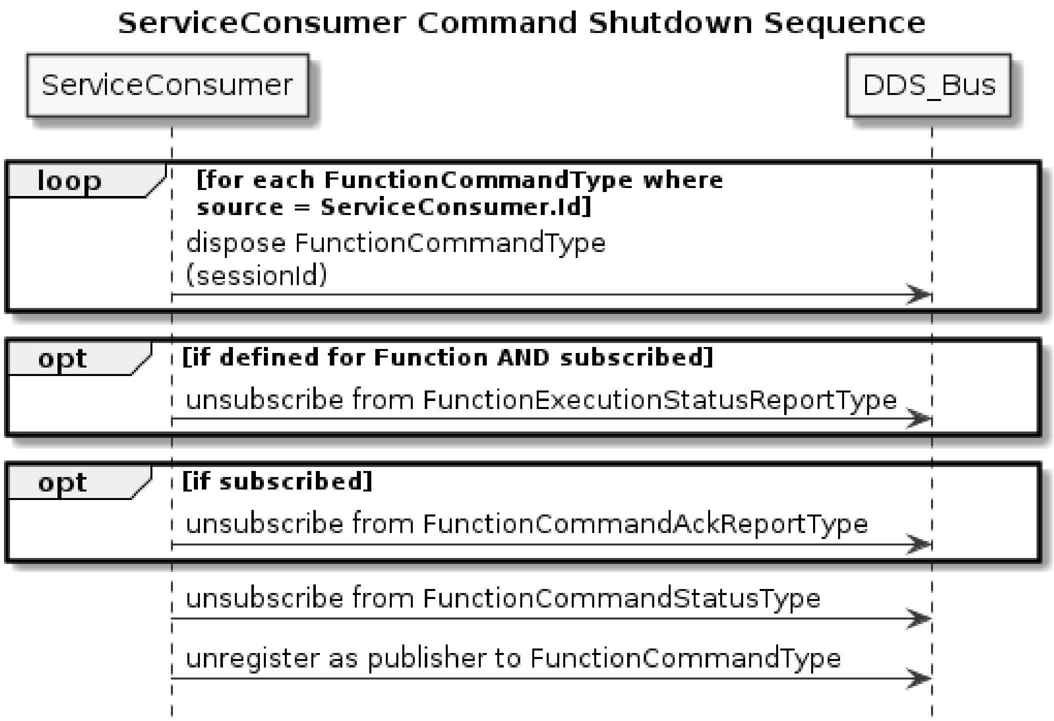

Figure 20: Sequence Diagram for Command Shutdown for Service Consumers.

4.2 Request / Reply

This section defines the flow of control for request/reply over the DDS bus. A request/reply is used to obtain data or status from a specific Service Provider.

A Service Provider is required to reply to all requests it receives. In the case of requests with no query data, this is accomplished via a DDS subscribe. In the case of a request with associated query data, a message with the query data must be published by the requester. To direct a request at a specific Service Provider or set of services, UMAA defines a `destination` GUID as part of requests.

The sequence diagrams in Sections 21 through 25 demonstrate different exchanges between a Service Consumer and Service Provider. Within the diagrams, the dashed arrows represent implementation-specific communications that are outside of UMAA's scope. Additionally, these sequence diagrams are examples of one possible implementation. Other implementations may have different communication patterns between the Service Provider and the Resource, or be implemented completely within the Service Provider process itself (no external Resource). However, in all implementations, UMAA-defined exchanges with the DDS bus between the Service Consumer and Service Provider must happen in the order shown within the sequence diagrams.

4.2.1 Request/Reply without Query Data

Figure 21 shows the sequence of exchanges in the case where there is no specific query data (i.e., the service is always just providing the current data to the bus).

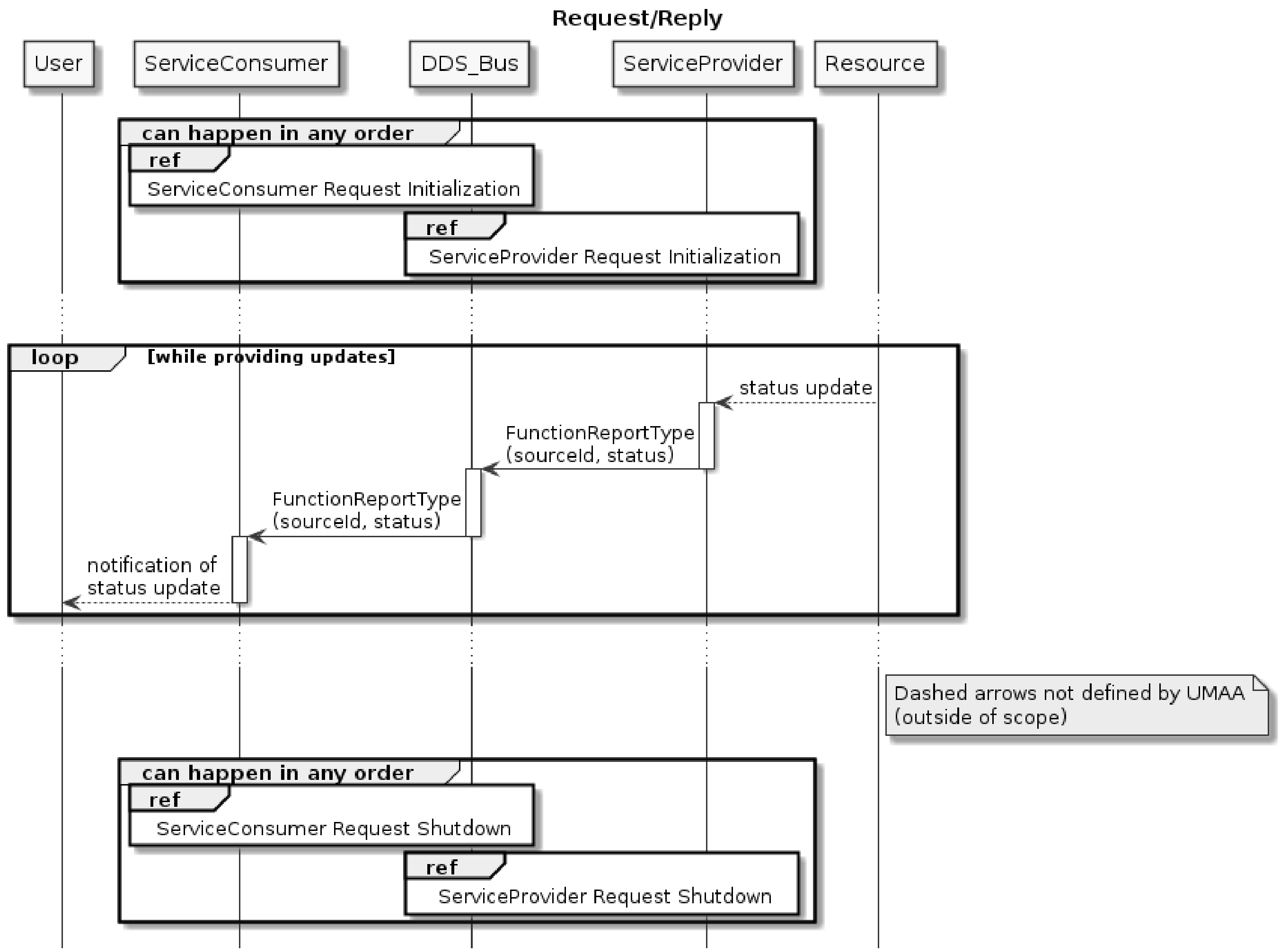

Figure 21: Sequence Diagram for a Request/Reply for Report Data That Does Not Require any Specific Query Data.

4.2.1.1 Service Provider Startup Sequence

The Service Provider registers as a publisher of `FunctionReportType`s to be able to respond to requests. The Service Provider must also handle reports that exist on the bus from a previous instantiation, either by providing an immediate update or, if the status is unrecoverable, disposing of the old `FunctionReportType`. This is shown in Figure 22.

As `FunctionReportType` updates are required (either through event-driven changes or periodic updates), the Service Provider publishes the updated data. The DDS bus will deliver the updates to the Service Consumer.

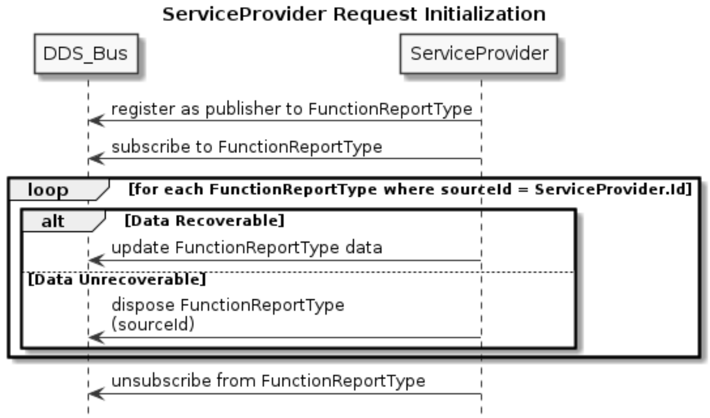

Figure 22: Sequence Diagram for Initialization of a Service Provider to Provide `FunctionReportTypes`.

4.2.1.2 Service Consumer Startup Sequence The Service Consumer subscribes to the `FunctionReportType` to signal an outstanding request for updates. This is shown in Figure 23.

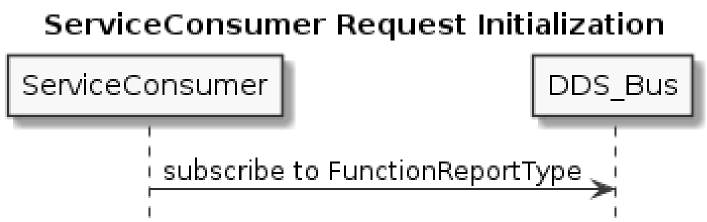

Figure 23: Sequence Diagram for Initialization of a Service Consumer to Request `FunctionReportTypes`.

4.2.1.3 Service Provider Shutdown To no longer provide `FunctionReportTypes`, the Service Provider disposes of the `FunctionReportType` and unregisters as a publisher of the data (shown in Figure 24).

Figure 24: Sequence Diagram for Shutdown of a Service Provider.

4.2.1.4 Service Consumer Shutdown To no longer request `FunctionReportTypes`, the Service Consumer unsubscribes from `FunctionReportType` (shown in Figure 25).

Figure 25: Sequence Diagram for Shutdown of a Service Consumer.

4.2.2 Request/Reply with Query Data

Currently, UMAA does not define any request/reply interactions with query data, but it is expected that some will be defined. When defined, this section will be expanded to describe how they must be used.

5 Support Operations (SO) Services and Interfaces

5.1 Services and Interfaces

The interfaces in the following subsections describe how each UCS-UMAA topic is defined by listing the name, namespace, and member attributes. The "name" corresponds with the message name of a given service interface. The "namespace" defines the scope of the "name" where similar commands are grouped together. The "member attributes" are fields that can be populated with differing data types, e.g. a generic "depth" attribute could be populated with a double data value. Note that using a UCS-UMAA "Topic Name" requires using the fully-qualified namespace plus the topic name.

Each interface topic is referenced by a UMAA service and is defined as either an input or output interface.

Attributes ending in one or more asterisk(s) denote the following:
* = Key (annotated with @key in IDL file; vendors may use different notation to indicate a key field)
† = Optional (annotated with @optional in IDL file; vendors may use different notation to indicate an optional field)

Optional fields should be handled as described in the UMAA Compliance Specification.

Commands issued on the DDS bus must be treated as if they are immutable in UMAA and, therefore, if updated (treated incorrectly as mutable), the resulting service actions are indeterminate and flow control protocols are no longer guaranteed.

Operations without DDS Topics

The following operations are all handled directly by DDS. They are marked in the operations tables with a ⊕.

query<...> - All query operations are used to retrieve the correlated report message. For UMAA, this operation is accomplished through subscribing to the appropriate DDS topic.

cancel<...> - All cancel operations are used to nullify the current command. For UMAA, this operation is accomplished through the DDS dispose action on the publisher.

report<...>CancelCommandStatus - All cancel reports are included here to show completeness of the MDE model mapping to UMAA. For UMAA, this operation is not used. Instead, the cancel status is inferred from the associated command status. If the cancel command is successful, the corresponding command will fail with a command status and reason of CANCELED. If the corresponding command status reports COMPLETED, then this cancel command has failed.

5.1.1 HealthReport

The purpose of this service is to provide health details.

Table 8: HealthReport Operations

Service Requests (Inputs)	Service Responses (Outputs)
queryHealth⊕	reportHealth

See Section 5.1 for an explanation of the inputs and outputs marked with a ⊕.

5.1.1.1 reportHealth

Description: This operation is used to report the most recent health status for each resource.

Namespace: UMAA::SO::HealthReport

Topic: HealthReport

Data Type: HealthReportType

Table 9: HealthReportType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAAStatus		
code	ErrorCodeEnumType	The types of system or subsystems associated with the error report.
logTime	DateTime	Log time when the error occurs.
severity	ErrorConditionEnumType	The type of error reported.
status	StringLongDescription	A detailed, human-readable string which specifies the status of the system or subsystem, such as the reason for failure. Systems should not parse or use any information from this for processing purposes.
resourceID*	NumericGUID	Unique Identifier of the health detail of the resource.

5.1.2 LogReport

The purpose of this service is to provide log messages.

Table 10: LogReport Operations

Service Requests (Inputs)	Service Responses (Outputs)
queryLogReport⊕	reportLogReport

See Section 5.1 for an explanation of the inputs and outputs marked with a ⊕.

5.1.2.1 reportLogReport

Description: This operation is used to report an entry on the bus that can be subscribed to and potentially entered into a system level log. This data is in addition to existing UMAA messages. It is a way of providing more detailed information than is normally published on the UMAA bus by the standard messaging.

Namespace: UMAA::SO::LogReport

Topic: LogReport

Data Type: LogReportType

Table 11: LogReportType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAAStatus		
entry	StringLongDescription	A human-readable description which specifies the contents of the log message. Systems should not parse or use any information from this for processing purposes.
level	LogLevelEnumType	Specifies the log level of the message.

5.1.3 ResourceAllocation

This service provides the interfaces necessary for attempting exclusive control of a resource, setting the priority ordering of resource consumers, and retrieving the current configuration and control information for a resource.

Resource Definition
For UMAA, a resource is defined as a logical grouping of UMAA service providers whose execution is mutually exclusive within the group. For example, when executing a GlobalWaypointControl command, a FinControl command cannot execute at the same time. A resource is identified by its resourceId, and the relationship of service providers to their resource is reported by the resource configuration message. Note that there can be different groupings of resources for different purposes. For example, an autopilot and its associated physical devices can be a viewed as a single resource for control, but each fin could be a resource for Built-In-Test (BIT).

Resource Consumer Definition
Resource consumers are simply consumers of UMAA services whose access is controlled by ResourceAllocation. A resource consumer is identified by the source attribute in the header of the intended command message. This identifier is used by ResourceAllocation when determining the consumer's priority (set using the priority command).

Configuring Resource Consumer Priority
Priority is configured at the resource level, based on resourceId. Priority is modelled using a sequence of resource consumer identifiers (see above), ordered by priority (low to high). Priority may be changed during runtime using the ResourceAllocation priority command, or it may be treated as static configuration. Updating the priority does not change the process in control - rather, it changes the priority state that will be used for the next command. In either case, the priority ordering for all resource consumers in the system must be configured before any resource allocation can take place. Similarly, if a resource consumer attempts a resource allocation and its identifier is not present in the priority configuration, the request must be rejected. Priority is primarily used internally by the ResourceAllocation service, but is published to the DDS bus for persistence. Note that service providers do not rely on this report data to determine whether a command can execute; rather, they use the ResourceAllocation control report to make this determination.

Additions to Flow Control - Resource Consumer
Implementing ResourceAllocation adds additional steps to the flow control for UMAA commands. Before sending a command to a service provider, the consumer must first command ResourceAllocation to attempt to allocate the resource for control. The consumer may only proceed with its service command only if it receives a ResourceAllocation command status of EXECUTING, indicating that it now has control of the resource. This process is detailed below:

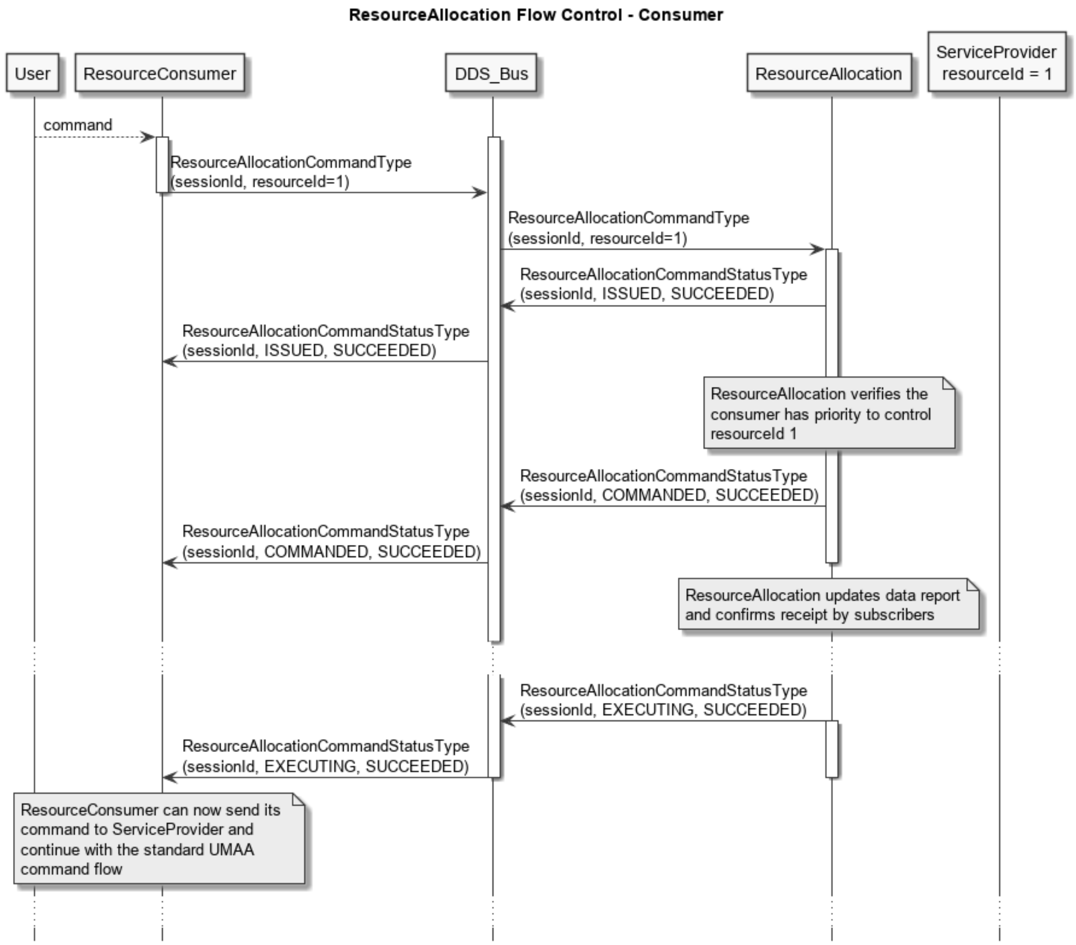

Figure 26: Sequence Diagram of Resource Consumer Requesting Control of a Resource

Additionally, resource consumers can set a duration when attempting control of a resource. If a duration is provided, the consumer must continue to ask for control within the timeout period so control allocation is not lost. This enables robust recovery for software failures and recovery.

Additions to Flow Control - Service Provider

Service providers must subscribe to the ResourceAllocation report. When a service provider receives a command, it uses this report to determine whether the consumer has control of the resource by checking the source of the incoming command message against the consumer currently in control. If the identifiers do not match, or the end time of the control session has elapsed, then the request must be rejected. This process is detailed below:

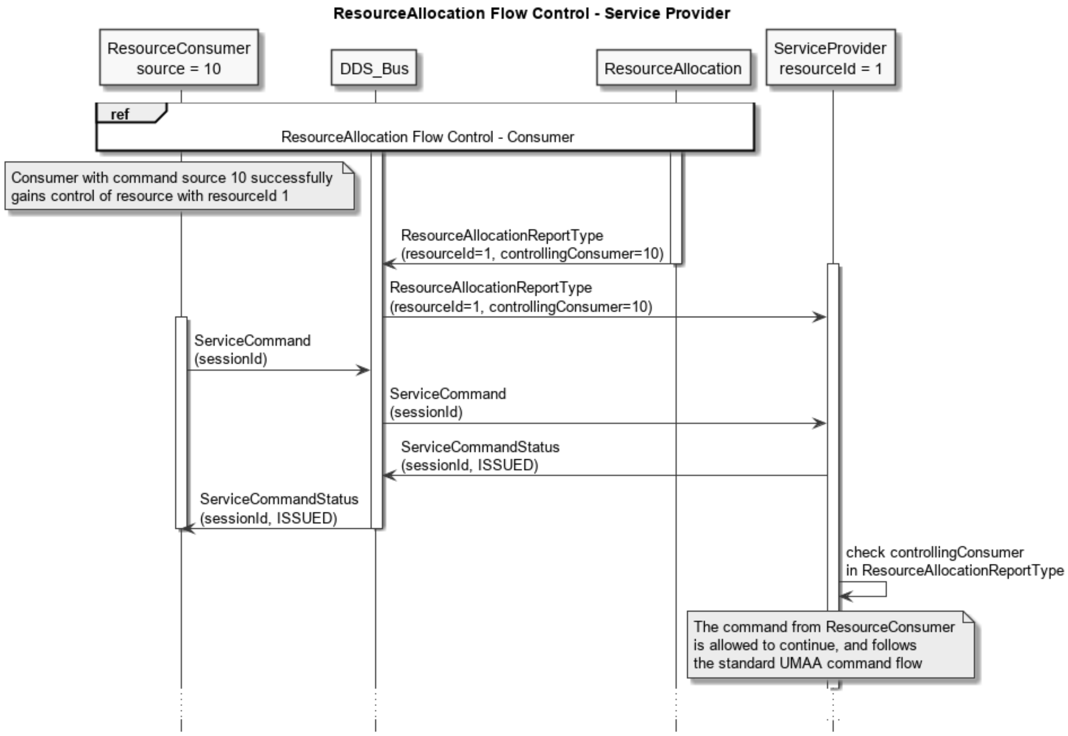

Figure 27: Sequence Diagram of Service Provider Verifying a Command using ResourceAllocation

Additions to Flow Control - Nested Service Provider

The ResourceAllocation command flow control has an additional step for nested services. A nested service provider is commanded by other service providers (and potentially by service consumers as well). For example, in the case where a GlobalWaypoint service implementation can achieve its functionality by sending a series of GlobalVector commands, the GlobalVector service provider is a nested service. When a nested service provider receives a command, it checks the configuration report to determine if the command was sent from another service in the same resource (using the command's source header field). If it was, the command can continue as normal, since the commanding service would have already followed the flow control above. If the command is received directly from the resource consumer originating the command, then the nested service must perform additional verification that the consumer has control of the resource. This process is detailed below:

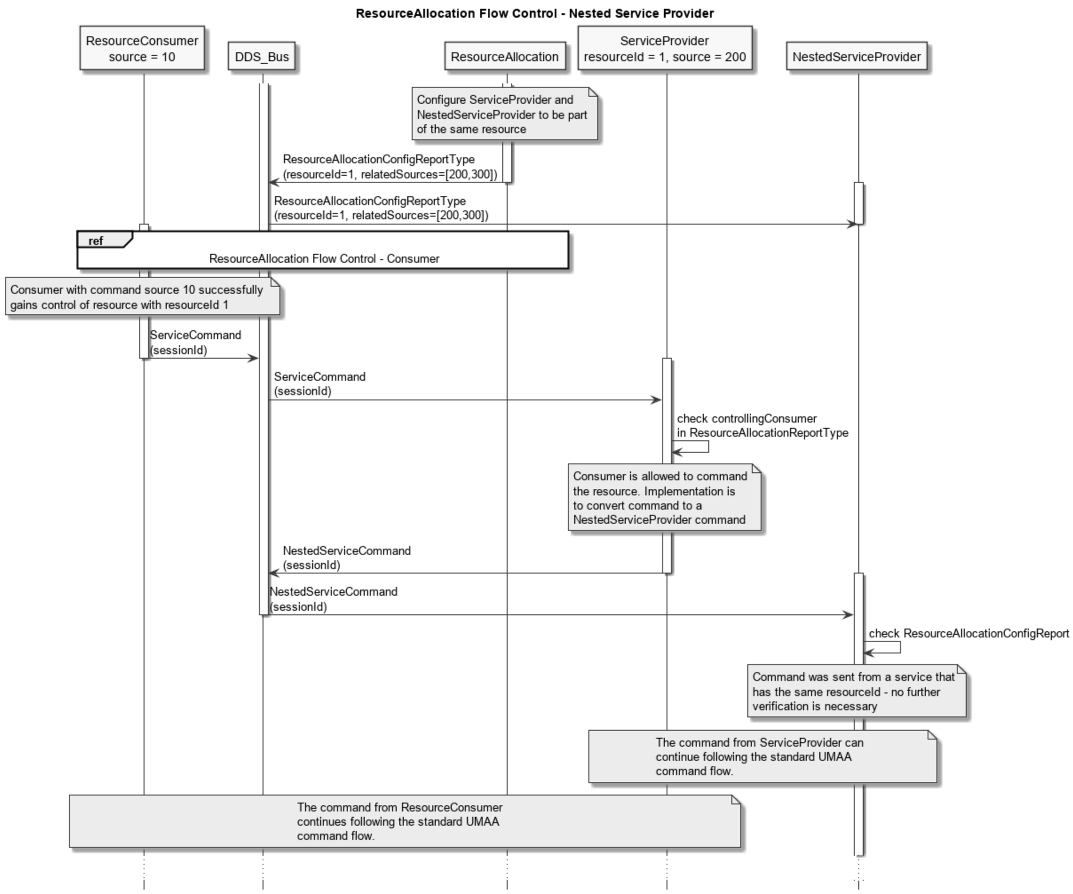

Figure 28: Sequence Diagram of Nested Service Provider Verifying a Command using ResourceAllocation

Ensuring Strong Data Consistency

To avoid report data race conditions, the ResourceAllocation service implementation must take additional steps to ensure strong consistency of ResourceAllocationReport data. Before allowing control of a resource, the ResourceAllocation service must ensure that all subscribers of ResourceAllocationReport have received the most recent sample. This means that service providers will have the data they need to verify the service consumer's command before it is sent. This consistency can be achieved in a number of ways, such as

- Using standard DDS capabilities by configuring the ResourceAllocation report DataWriter to be reliable and using wait_for_acknowledgements()
- Using implementation-specific DDS extensions
- Implementing multiple ResourceAllocation services and using a combination of flow control and other methods above to communicate across a proxy (see examples online)

Avoiding Pitfalls

The ResourceAllocation service exists to deconflict requests coming from multiple service consumers, which would otherwise cause a fight over a particular resource. There is no mechanism in place to prevent a single consumer who has gained control of a resource to issue concurrent commands to multiple service providers within the resource. Doing so is bad engineering practice and should be avoided to ensure command determinism.

Table 12: ResourceAllocation Operations

Service Requests (Inputs)	Service Responses (Outputs)
setResourceAllocation	reportResourceAllocationCommandStatus
queryResourceAllocationCommandAck⊕	reportResourceAllocationCommandAck
cancelResourceAllocationCommand⊕	reportResourceAllocationCancelCommandStatus⊕

Service Requests (Inputs)	Service Responses (Outputs)
queryResourceAllocation⊕	reportResourceAllocation
queryResourceAllocationConfig⊕	reportResourceAllocationConfig
setResourceAllocationPriority	reportResourceAllocationPriorityCommandStatus
queryResourceAllocationPriorityCommandAck⊕	reportResourceAllocationPriorityCommandAck
cancelResourceAllocationPriorityCommand⊕	reportResourceAllocationPriorityCancelCommandStatus⊕
queryResourceAllocationPriority⊕	reportResourceAllocationPriority

See Section 5.1 for an explanation of the inputs and outputs marked with a ⊕.

5.1.3.1 reportResourceAllocation

Description: This operation is used to report the current resources and what consumer currently owns each resource.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationReport

Data Type: ResourceAllocationReportType

Table 13: ResourceAllocationReportType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAAStatus		
controlInfo	sequence<ResourceAllocationControlInfo> max size = 256	A list of the control information for every resource defined by ResourceAllocation.

5.1.3.2 reportResourceAllocationCommandAck

Description: This operation is used to provide the ResourceAllocation commanded values.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationCommandAckReport

Data Type: ResourceAllocationCommandAckReportType

Table 14: ResourceAllocationCommandAckReportType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAACommandStatusBase		
command	ResourceAllocationCommandType	The source command.

5.1.3.3 reportResourceAllocationCommandStatus

Description: This operation is used to report the status of the current resource allocation command.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationCommandStatus

Data Type: ResourceAllocationCommandStatusType

Table 15: ResourceAllocationCommandStatusType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAACommandStatus		

5.1.3.4 reportResourceAllocationConfig

Description: This operation is used to report all service provider identifiers that are sharing a single resource.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationConfigReport

Data Type: ResourceAllocationConfigReportType

Table 16: ResourceAllocationConfigReportType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAAStatus		
resources	sequence<ResourceAllocationDefinitionType> max size = 256	The configuration of each resource.

5.1.3.5 reportResourceAllocationPriority

Description: This operation is used to report the current priority ordering of resource consumers.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationPriorityReport

Data Type: ResourceAllocationPriorityReportType

Table 17: ResourceAllocationPriorityReportType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAAStatus		
priorities	sequence<ResourceAllocationPriorityInfo> max size = 32	The priority ordering of resource consumers for each resource defined by ResourceAllocation.

5.1.3.6 reportResourceAllocationPriorityCommandAck

Description: This operation is used to provide the ResourceAllocationPriority commanded values.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationPriorityCommandAckReport

Data Type: ResourceAllocationPriorityCommandAckReportType

Table 18: ResourceAllocationPriorityCommandAckReportType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAACommandStatusBase		
command	ResourceAllocationPriorityCommandType	The source command.

5.1.3.7 reportResourceAllocationPriorityCommandStatus

Description: This operation is used to report the status of the current priority command.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationPriorityCommandStatus

Data Type: ResourceAllocationPriorityCommandStatusType

Table 19: ResourceAllocationPriorityCommandStatusType Message Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAACommandStatus		

5.1.3.8 setResourceAllocation

Description: This operation is used to set the current resource allocation command. The source attribute in the header of this command is used to identify the consumer that is requesting the resource to be allocated.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationCommand

Data Type: ResourceAllocationCommandType

Table 20: ResourceAllocationCommandType Message Definition

Attribute Name	Attribute Type	Attribute Description
	Additional fields included from	UMAA::UMAACommand
duration†	DurationSeconds	Specifies the end of the valid time period for the control session. Once the control session ends, the resource will become available for other requesters to control. If this field is empty, then the duration is assumed to be infinite.
resourceID	NumericGUID	The identifier of the resource to control.

5.1.3.9 setResourceAllocationPriority

Description: This operation is used to set the ordered priority of consumers who will be requesting access. All potential consumers must be on this list to be enabled to request control allocation.

Namespace: UMAA::SO::ResourceAllocation

Topic: ResourceAllocationPriorityCommand

Data Type: ResourceAllocationPriorityCommandType

Table 21: ResourceAllocationPriorityCommandType Message Definition

Attribute Name	Attribute Type	Attribute Description
	Additional fields included from	UMAA::UMAACommand
priorities	sequence<NumericGUID> max size = 100	The priority-ordered (low to high) sequence of resource consumer source identifiers.
resourceID	NumericGUID	The identifier of the resource being set.

5.2 Common Data Types

Common data types define DDS types that are referenced throughout the UMAA model. These DDS types are considered common because they can be re-used as the data type for many attributes defined in service interface topics, interface topics, and other common data types. These data types are not intended to be directly published to/subscribed as DDS topics.

5.2.1 UCSMDEInterfaceSet

Namespace: UMAA::UCSMDEInterfaceSet

Description: Defines the common UCSMDE Interface Set Message Fields.

Table 22: UCSMDEInterfaceSet Structure Definition

Attribute Name	Attribute Type	Attribute Description
timeStamp	DateTime	The time at which the data is valid.

5.2.2 UMAACommand

Namespace: UMAA::UMAACommand

Description: Defines the common UMAA Command Message Fields.

Table 23: UMAACommand Structure Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UCSMDEInterfaceSet		
source*	NumericGUID	The unique identifier of the originating source of the command interface.
destination*	NumericGUID	The unique identifier of the destination of the command interface.
sessionID*	NumericGUID	The identifier of the session.

5.2.3 UMAAStatus

Namespace: UMAA::UMAAStatus

Description: Defines the common UMAA Status Message Fields.

Table 24: UMAAStatus Structure Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UCSMDEInterfaceSet		
source*	NumericGUID	The unique identifier of the originating source of the status interface.

5.2.4 UMAACommandStatusBase

Namespace: UMAA::UMAACommandStatusBase

Description: Defines the common UMAA Command Status Base Message Fields.

Table 25: UMAACommandStatusBase Structure Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UCSMDEInterfaceSet		
source*	NumericGUID	The unique identifier of the originating source of the command status interface.
sessionID*	NumericGUID	The identifier of the session.

5.2.5 UMAACommandStatus

Namespace: UMAA::UMAACommandStatus

Description: Defines the common UMAA Command Status Message Fields.

Table 26: UMAACommandStatus Structure Definition

Attribute Name	Attribute Type	Attribute Description
Additional fields included from UMAA::UMAACommandStatusBase		
commandStatus	CommandStatusEnumType	The status of the command.
commandStatusReason	CommandStatusReasonEnumType	The reason for the status of the command.
logMessage	StringLongDescription	Human-readable description related to response. Systems should not parse or use any information from this for processing purposes.

5.2.6 DateTime

Namespace: UMAA::Measurement::DateTime

Description: Describes an absolute time. Conforms with POSIX time standard (IEEE Std 1003.1-2017) epoch reference point of January 1st, 1970 00:00:00 UTC.

Table 27: DateTime Structure Definition

Attribute Name	Attribute Type	Attribute Description
seconds	DateTimeSeconds	The number of seconds offset from the standard POSIX (IEEE Std 1003.1-2017) epoch reference point of January 1st, 1970 00:00:00 UTC.
nanoseconds	DateTimeNanoSeconds	The number of nanoseconds elapsed within the current DateTimeSecond.

5.2.7 Quaternion

Namespace: BasicTypes::Quaternion

Description: Defines a four-element vector that can be used to encode any rotation in a 3D coordinate system.

Table 28: Quaternion Structure Definition

Attribute Name	Attribute Type	Attribute Description
a		Real number a.
b		Real number b.
c		Real number c.
d		Real number d.

5.2.8 ResourceAllocationControlInfo

Namespace: UMAA::SO::ResourceAllocation::ResourceAllocationControlInfo

Description: This structure is used to define attributes related to the controller of a resource.

Table 29: ResourceAllocationControlInfo Structure Definition

Attribute Name	Attribute Type	Attribute Description
controlSession†	ResourceAllocationControlSession	Information on the consumer currently controlling the resource. If empty, this resource is not currently allocated for control.
resourceID*	NumericGUID	The identifier of the resource being controlled.

5.2.9 ResourceAllocationControlSession

Namespace: UMAA::SO::ResourceAllocation::ResourceAllocationControlSession

Description: This structure is used to define attributes related to the current controller of a resource.

Table 30: ResourceAllocationControlSession Structure Definition

Attribute Name	Attribute Type	Attribute Description
controllingConsumer	NumericGUID	The source identifier of the resource consumer in control of the resource.
endTime†	DateTime	The absolute end time of the consumer's control. After this time is reached, the resource is available to be controlled by another process. If this field is empty, then the duration is assumed to be infinite.

5.2.10 ResourceAllocationDefinitionType

Namespace: UMAA::SO::ResourceAllocation::ResourceAllocationDefinitionType

Description: This structure is used to define the attributes associated with a resource - that is, a collection of related service providers whose functionality cannot be executed simultaneously.

Table 31: ResourceAllocationDefinitionType Structure Definition

Attribute Name	Attribute Type	Attribute Description
relatedSources	sequence<NumericGUID> max size = 100	The source identifiers of each service that is logically part of this resource. For instance, this resource could represent driving-related services at large. This field would then contain the source of each driving-related service provider active in the system.
resourceID*	NumericGUID	The identifier of the resource being controlled.

5.2.11 ResourceAllocationPriorityInfo

Namespace: UMAA::SO::ResourceAllocation::ResourceAllocationPriorityInfo

Description: This structure is used to define the configuration of resource control priority for a particular resource.

Table 32: ResourceAllocationPriorityInfo Structure Definition

Attribute Name	Attribute Type	Attribute Description
priorities	sequence<NumericGUID> max size = 100	The priority-ordered (low to high) sequence of client identifiers.
resourceID*	NumericGUID	The identifier of the resource being controlled.

5.3 Enumerations

Enumerations are used extensively throughout UMAA. This section lists the values associated with each enumeration defined in UCS-UMAA.

5.3.1 CommandStatusReasonEnumType

Namespace: UMAA::Common::MaritimeEnumeration::CommandStatusReasonEnumType

Description: Defines a mutually exclusive set of reasons why a command status state transition has occurred.

Table 33: CommandStatusReasonEnumType Enumeration

Enumeration Value	Description
CANCELED	Indicates a transition to the CANCELED state when the command is canceled successfully.
INTERRUPTED	Indicates a transition to the FAILED state when the command has been interrupted by a higher priority process.
OBJECTIVE_FAILED	Indicates a transition to the FAILED state when the commanded resource is unable to achieve the command's objective due to external factors.
RESOURCE_FAILED	Indicates a transition to the FAILED state when the commanded resource is unable to achieve the command's objective due to resource or platform failure.
RESOURCE_REJECTED	Indicates a transition to the FAILED state when the commanded resource rejects the command for some reason.
SERVICE_FAILED	Indicates a transition to the FAILED state when the commanded resource is unable to achieve the command's objective due to processing failure.
SUCCEEDED	Indicates the conditions to proceed to this state have been met and a normal state transition has occurred.
TIMEOUT	Indicates a transition to the FAILED state when the command is not acknowledged within some defined time bound.
UPDATED	Indicates a transition back to the ISSUED state from a non-terminal state when the command has been updated.
VALIDATION_FAILED	Indicates a transition to the FAILED state when the command contains missing, out-of-bounds, or otherwise invalid parameters.

5.3.2 ErrorCodeEnumType

Namespace: UMAA::Common::MaritimeEnumeration::ErrorCodeEnumType

Description: A mutually exclusive set of values that defines the error codes.

Table 34: ErrorCodeEnumType Enumeration

Enumeration Value	Description
ACTUATOR	Actuator
FILESYS	File system
NONE	None
POWER	Power
PROCESSOR	Processor
RAM	RAM
ROM	ROM

Enumeration Value	Description
SENSOR	Sensor
SOFTWARE	Software

5.3.3 ErrorConditionEnumType

Namespace: UMAA::Common::MaritimeEnumeration::ErrorConditionEnumType

Description: A mutually exclusive set of values that defines the error condition.

Table 35: ErrorConditionEnumType Enumeration

Enumeration Value	Description
ERROR	An error condition is reported and expected to seriously compromise use of the reporting component or device.
FAIL	An error condition is reported with severity indicating component or device failure.
INFO	An error condition is reported, but impact on operation and performance is minimal.
NONE	No error condition exists.
WARN	An error condition is reported and expected to have significant impact on component or device performance.

5.3.4 LogLevelEnumType

Namespace: UMAA::Common::MaritimeEnumeration::LogLevelEnumType

Description: Defines a mutually exclusive set of values that defines the log level.

Table 36: LogLevelEnumType Enumeration

Enumeration Value	Description
ERROR	An error message level.
INFORMATION	An informational message level.
WARNING	A warning message level.

5.3.5 CommandStatusEnumType

Namespace: UMAA::Common::MaritimeEnumeration::CommandStatusEnumType

Description: Defines a mutually exclusive set of values that defines the states of a command as it progresses towards completion.

Table 37: CommandStatusEnumType Enumeration

Enumeration Value	Description
CANCELED	The command was canceled by the requestor before the command completed successfully.
COMMANDED	The command has been placed in the resource's command queue but has not yet been accepted.
COMPLETED	The command has been completed successfully.
EXECUTING	The command is being performed by the resource and has not yet been completed.
FAILED	The command has been attempted, but was not successful.
ISSUED	The command has been issued to the resource (typically a sensor or streaming device), but processing has not yet commenced.

5.4 Type Definitions

This section describes the type definitions for UMAA. The table below lists how UMAA defined types are mapped to the DDS primitive types.

Table 38: Type Definitions

Type Name	Primitive Type	Range of Values	Description
DateTimeNanoseconds	long	units=Nanoseconds minInclusive=0 maxInclusive=999999999 fractionDigits=0	number of nanoseconds elapsed within the current second.
DateTimeSeconds	longlong	units=Seconds minInclusive=-9223372036854775807 maxInclusive=9223372036854775807 fractionDigits=0	seconds offset from the standard POSIX (IEEE Std 1003.1-2017) epoch reference point of January 1st, 1970 00:00:00 UTC.
DurationSeconds	double	fractionDigits=6 maxInclusive=37817280 minInclusive=0 units=Seconds referenceFrame=Counting	Represents a time duration in seconds.
LargeCollectionSize	long	fractionDigits=0 maxInclusive=2147483647 minInclusive=0 units=N/A	Specifies the size of a Large Collection.
NumericGUID	octet[16]	units=N/A minInclusive=0 maxInclusive=(2^128)-1 fractionDigits=0	Represents a 128-bit number according to RFC 4122 variant 2.
StringLongDescription	string	length=4095 units=N/A minInclusive=N/A maxInclusive=N/A	Represents a long format description.

A Appendices

A.1 Glossary

Note: This glossary aims to define terms that are uncommon, or have a special meaning in the context of UMAA and/or the DoD. This glossary covers the complete UMAA specification. Not every word defined here appears in every ICD.

Almanac Data (GPS)	A navigation message that contains information about the time and status of the entire satellite constellation.
Coulomb	The SI unit of electric charge, equal to the quantity of electricity conveyed in one second by a current of one ampere.
Ephemeris Data (GPS)	A navigation message used to calculate the position of each satellite in orbit.
Glowplug or Glow Plug	A heating device used to aid in starting diesel engines.
Interoperability	1) The ability to act together coherently, effectively, and efficiently to achieve tactical, operational, and strategic objectives. 2) The condition achieved among communications-electronics systems or items of communications-electronics equipment when information or services can be exchanged directly and satisfactorily between them and/or their users.
Mean Sea Level	The average height of the surface of the sea for all stages of the tide; used as a reference for elevations.
Middleware	A type of computer software that provides services to software applications beyond those available from the operating system. Middleware makes it easier for software developers to implement communication and input/output, so they can focus on the specific purpose of their application.
SoaML	The Service oriented architecture Modeling Language (SoaML) specification that provides a metamodel and a UML profile for the specification and design of services within a service-oriented architecture. The specification is managed by the Object Management Group (OMG).

A.2 Acronyms

Note: This acronym list is included in every ICD and covers the complete UMAA specification. Not every acronym appears in every ICD.

ADD	Architecture Design Description
AGL	Above Sea Level
ASF	Above Sea Floor
BSL	Below Sea Level
BWL	Beam at Waterline
C2	Command and Control
CMD	Command
CO	Comms Operations
CPA	Closest Point of Approach
CTD	Conductivity, Temperature and Depth
DDS	Data Distribution Service
DTED	Digital Terrain Elevation Data
EGM	Earth Gravity Model
EO	Engineering Operations
FB	Feedback
GUID	Globally Unique Identifier
HM&E	Hull, Mechanical, & Electrical
ICD	Interface Control Document

ID	Identifier
IDL	Interface Definition Language Specification
IMO	International Maritime Organization
INU	Inertial Navigation Unit
LDM	Logical Data Model
LOA	Length Over All
LRC	Long Range Cruise
LWL	Length at Waterline
MDE	Maritime Domain Extensions
MEC	Maximum Endurance Cruise
MM	Mission Management
MMSI	Maritime Mobile Service Identity
MO	Maneuver Operations
MRC	Maximum Range Cruise
MSL	Mean Sea Level
OMG	Object Management Group
PIM	Platform Independent Model
PMC	Primary Mission Control
PNT	Precision Navigation and Timing
PO	Processing Operations
PSM	Platform Specific Model
RMS	Root-Mean-Square
RPM	Revolutions per minute
RTPS	Real Time Publish Subscribe
RTSP	Real Time Streaming Protocol
SA	Situational Awareness
SEM	Sensor and Effector Management
SO	Support Operations
SoaML	Service-oriented architecture Modeling Language
STP	Standard Temperature and Pressure
UCS	Unmanned Systems Control Segment
UMAA	Unmanned Maritime Autonomy Architecture
UML	Unified Modeling Language
UMS	Unmanned Maritime System
UMV	Unmanned Maritime Vehicle
UxS	Unmanned System
WGS84	Global Coordinate System
WMM	World Magnetic Model
WMO	World Meteorological Organization