ORNL/TM-2023/3171

INTERSECT Architecture Specification: Microservice Architecture (Version 0.9)

Michael J. Brim
Christian Engelmann

September 30, 2023

**Approved for public release.
Distribution is unlimited.**

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2023/3171

Laboratory Directed Research and Development Program
Self-Driven Experiments for Science/Interconnected Science Ecosystem (INTERSECT) Initiative

INTERSECT Architecture Specification: Microservice Architecture (Version 0.9)

Michael J. Brim, Christian Engelmann

September 30, 2023

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES . . . v
LIST OF TABLES . . . vii
ACRONYMS AND ABBREVIATIONS . . . ix
INTERSECT TERMINOLOGY . . . xi
ACKNOWLEDGEMENTS . . . xiii
ABSTRACT . . . xv
REVISION RECORD . . . xvii
1 INTRODUCTION . . . 1
2 INTERSECT MICROSERVICE ARCHITECTURE . . . 2
2.1 INTRODUCTION TO MICROSERVICES ARCHITECTURE . . . 2
2.2 MICROSERVICES ARCHITECTURE IN INTERSECT . . . 3
2.3 COMMONALITIES OF INTERSECT MICROSERVICES . . . 4
3 CLASSIFICATION OF INTERSECT MICROSERVICES . . . 6
3.1 INTERSECT MICROSERVICE CAPABILITY DEFINITION FORMAT . . . 6
3.2 INTERSECT INFRASTRUCTURE MICROSERVICE CAPABILITIES . . . 8
3.2.1 General Utility . . . 9
3.2.1.1 Capability: *Availability Status* . . . 10
3.2.1.2 Capability: *Controller Status* . . . 11
3.2.1.3 Capability: *Parameter Configuration* . . . 13
3.2.1.4 Capability: *UUID Generation* . . . 15
3.2.2 INTERSECT Data and Information Management . . . 15
3.2.2.1 Data Storage Services . . . 15
3.2.2.2 Data Transfer Services . . . 16
3.2.2.3 Database and Information Management Services . . . 16
3.2.2.4 A Data Model for INTERSECT Microservices . . . 16
3.2.2.5 Capability: *Entity-Relationship Catalog* . . . 18
3.2.2.6 Capability: *Data Catalog* . . . 23
3.2.2.7 Capability: *Data Storage* . . . 27
3.2.2.8 Capability: *Data Stream* . . . 34
3.2.2.9 Capability: *Data Transfer Orchestration* . . . 39
3.2.2.10 Capability: *Data Transfer Endpoint* . . . 41
3.2.3 INTERSECT System Management . . . 44
3.2.3.1 Capability: *System Information Catalog* . . . 46
3.2.3.2 Capability: *System Manager* . . . 51
3.2.3.3 Capability: *Systems Registrar* . . . 56
3.2.4 INTERSECT Resource Adapters . . . 60
3.2.4.1 Compute Adapters . . . 60

3.2.4.2 Capability: *Compute Allocation* . . . 61
3.2.4.3 Capability: *Compute Queue* . . . 65
3.2.4.4 Capability: *Compute Queue Reservation* . . . 68
3.2.4.5 Capability: *Application Execution* . . . 71
3.2.4.6 Capability: *Container Execution* . . . 73
3.2.4.7 Capability: *Host Command Execution* . . . 76
3.3 EXPERIMENT-SPECIFIC MICROSERVICE CAPABILITIES . . . 77
3.3.1 Experiment Control Microservices . . . 77
3.3.1.1 Instrument Adapters . . . 77
3.3.1.2 Capability: *Instrument Controller* . . . 78
3.3.2 Experiment Data Microservices . . . 81
3.3.3 Experiment Design Microservices . . . 81
4 ORCHESTRATION AND DEPLOYMENT OF INTERSECT MICROSERVICES . . . 82
4.1 MICROSERVICE ORCHESTRATION DESIGN PATTERNS . . . 82
4.1.1 Asynchronous Messaging vs. RESTful Services . . . 82
4.1.2 Conductor vs. Choreography . . . 83
4.2 MICROSERVICE DEPLOYMENT DESIGN PATTERNS . . . 84
4.2.1 Sidecar Pattern . . . 85
4.2.2 Ambassador Proxy Pattern . . . 86
4.2.3 Service Mesh Pattern . . . 86
REFERENCES . . . 89
APPENDICES . . . 91
A INTERSECT INFRASTRUCTURE MICROSERVICE INTERACTION SEQUENCES . . . 91
A-1 Registration of INTERSECT Systems, Services, and Resources . . . 91
A-2 Running Applications using On-Demand or Batch Computing Resources . . . 96
A-3 Data Management for Application Data Products . . . 103

LIST OF FIGURES

2-1 Potential Classes of INTERSECT Microservices 3
2-2 Interaction Patterns for INTERSECT Microservices 4
3-1 Microservice Capability Definition Format 7
3-2 INTERSECT Data Model 17
3-3 INTERSECT System, Subsystem, Service, and Resource Hierarchy 44
3-4 INTERSECT System Entity-Relation Model 45
4-1 Command Interaction Pattern for Client-Server and Asynchronous Messaging 83
4-2 Request-Reply Interaction Pattern for Client-Server and Asynchronous Messaging 84
4-3 Asynchronous Status or Event Interaction Pattern for Client-Server and Asynchronous Messaging 85
4-4 Sidecar Pattern 86
4-5 Ambassador Proxy Pattern 87
4-6 Service Mesh Pattern 88
A-1 Example interaction sequence showing common components. 92
A-2 Sequence for registering an INTERSECT system. 93
A-3 Sequence for registering INTERSECT system resources. 94
A-4 Sequence for registering an INTERSECT system service. 95
A-5 Sequence for running an application within an on-demand allocation of computing resources. 97
A-6 Sequence for allocating and releasing computing resources. 98
A-7 Sequence for running and monitoring a program within allocated computing resources. . . . 99
A-8 Sequence for running an application within a batch computing job. 100
A-9 Sequence for running and monitoring a batch compute job. 101
A-10 Sequence for discovery of batch computing queues. 102
A-11 Sequence for an application to manage a data namespace. 104
A-12 Sequence for an application to manage a data collection. 105
A-13 Sequence for an application to manage its data items. 106
A-14 Sequence for an application to publish data products. 107
A-15 Sequence for negotiating data transfers between INTERSECT systems. 108
A-16 Sequence for transferring a data product between INTERSECT systems. 109
A-17 Sequence for transferring a data item between INTERSECT systems. 110

LIST OF TABLES

3-1 Generic Data Types and Structures and Associated Schema-specific Representations 8

ACRONYMS AND ABBREVIATIONS

ACID atomic, consistent, idempotent, and durable. 17
AI artificial intelligence. xv, 1, 60
API application programming interface. 3
DOE U. S. Department of Energy. 1
E-R Entity-Relationship. 17, 18
FTP File Transfer Protocol. 16
GPFS General Parallel File System. 15
HDD hard disk drive. 16
HPC high-performance computing. 44, 60
HPSS High Performance Storage System. 15
HTTP Hypertext Transfer Protocol. 3, 16, 82
HTTPS Hypertext Transfer Protocol Secure. 16
INTERSECT Self-driven Experiments for Science / Interconnected Science Ecosystem. xv
LSF IBM Spectrum Load Sharing Facility. 60
NFS Network File System. 15
NVM non-volatile memory. 16
ORNL Oak Ridge National Laboratory. xv, 1
PBS Portable Batch System. 60
REST Representational State Transfer. 3
SLURM Simple Linux Utility for Resource Management. 60
SME subject matter expert. 2
SoS system of systems. xv, 1, 3
SSL Secure Sockets Layer. 16
UUID universally unique identifier. 15

INTERSECT TERMINOLOGY

Term	Definition
Activity	Work, not specific to a single organization, weapon system or individual that transforms inputs (Resources) into outputs (Resources) or changes their state. [DODAF]
Administrator	Role of entity that maintains one or more systems; complete view of "their" system (their jurisdiction/domain/realm/area); Limited to a given jurisdiction (i.e., their administrative domain). [IAT]
Administrative Domain	Jurisdiction of control. [IAT]
Application	A computer process, or a set of coordinating computer processes, that perform activities to accomplish one or more predetermined goals. [IAT]
Campaign	A scientific endeavor that may consist of one or more Experiments that may take place sequentially or in parallel to answer a broader overarching scientific question. [IAT]
Capability	The ability to achieve a desired effect under specified [performance] standards and conditions through combinations of ways and means [activities and resources] to perform a set of activities. [DODAF]
Experiment	An indivisible component of a scientific endeavor that typically involves measurements, computation, and/or data analysis. An experiment is performed with a unique set of conditions and/or parameters. So long as the parameters are feasible, every Experiment will have a clear start and a predicable end. Insights of an experiment are often narrow and may not answer broader scientific questions. [IAT]
Facility	A real property entity consisting of underlying land and one or more of the following: a building, a structure (including linear structures), a utility system, or pavement. [DODAF]
Institution	See Organization.
I-System	Abbreviation for an "INTERSECT System". [IAT]
Jurisdiction	Realm of authority; Administrative domain. [IAT]
Maintainer	See Operator.
Microservice	A service that is designed according to the Microservices Architecture methodology and that implements a Microservice Contract for its clients. [IAT]
Microservice Architecture	A design methodology for structuring a distributed Application as a networked collection of loosely-coupled services that are independently developed, maintained, and operated. [IAT]
Microservice Function	A specific Microservice functionality with clearly defined input and output data. The function may have associated service state pre-conditions and/or post-conditions. [IAT]
Microservice Contract	The complete set of Microservice Functions provided by a Microservice to its clients, typically scoped using domain-driven design and defined using an API. [IAT]
Operator	Role of entity that maintains one or more resources; different view of system (i.e., in contrast to User). [IAT]
Organization	A specific real-world assemblage of people and other resources organized for an on-going purpose. [DODAF]

Owner	Role of entity fiscally responsible for a resource; Vested interest; Possibly approver for a resource. [IAT]
Performer	Any entity - human, automated, or any aggregation of human and/or automated - that performs an activity and provides a capability. [DODAF]
PersonType	A category of persons defined by the role or roles they share that are relevant to an architecture. [DODAF]
Provider	Role of entity that manufactures a resource (intention is distinction between provider/owner), e.g., SecDevOps is provider of SDK. Provider creates the Service that the Operator maintains. Provider creates the Instrument that the Operator maintains. [IAT]
Resource	Data, Information, Performers, Materiel, or Personnel Types that are produced or consumed. [DODAF]
Role	Performs a specific function; implies access rights for resources; checks for adherence to resource/jurisdiction/facility polices. [IAT]
Service	A mechanism to enable access to a set of one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. The mechanism is a Performer. The "capabilities" accessed are Resources – Information, Data, Materiel, Performers, and Geo-political Extents. [DODAF]
Sub-system[1]	A self-contained system within a larger system that is capable of both independent operation as well as coordinated interaction with other systems. [IAT]
Sub-system[2]	(Alt. definition) A self-contained system within a larger integrated System. [IAT]
System (or I-System)	A logical entity with operational and managerial independence that provides utility to the overall System of Systems. A System may utilize one or more physical resources and may be geographically distributed. Systems communicate with each other with the INTERSECT protocol for control purposes. A System provides this utility via one or more Services. A System may have Sub-systems. [IAT]
User	Role for entity using the system (not responsible for administration). [IAT]
Workflow	Activities that are performed according to a recipe (i.e. sequential, in a DAG) or script. Static or Dynamic Workflow. [IAT]

Sources

[DoDAF]	DoD Architecture Framework v2.02 Glossary, https://dodcio.defense.gov/Library/DoD-Architecture-Framework/dodaf20_glossary
[IAT]	INTERSECT Architecture Team

ACKNOWLEDGEMENTS

Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy.

ABSTRACT

Oak Ridge National Laboratory (ORNL)'s Self-driven Experiments for Science / Interconnected Science Ecosystem (INTERSECT) architecture project, titled "An Open Federated Architecture for the Laboratory of the Future", creates an open federated hardware/software architecture for the laboratory of the future using a novel system of systems (SoS) and microservice architecture approach, connecting scientific instruments, robot-controlled laboratories and edge/center computing/data resources to enable autonomous experiments, "self-driving" laboratories, smart manufacturing, and artificial intelligence (AI)-driven design, discovery and evaluation. The project describes science use cases as design patterns that identify and abstract the involved hardware/software components and their interactions in terms of control, work and data flow. It creates a SoS architecture of the federated hardware/software ecosystem that clarifies terms, architectural elements, the interactions between them and compliance. It further designs a federated microservice architecture, mapping science use case design patterns to the SoS architecture with loosely coupled microservices, standardized interfaces and multi programming language support. The primary deliverable of this project is an INTERSECT Open Architecture Specification, containing the science use case design pattern catalog, the federated SoS architecture specification and the microservice architecture specification. This document represents the microservice architecture specification of the INTERSECT Open Architecture Specification.

REVISION RECORD

Version	Date	Description
0.9	09/30/2023	Public release with the following changes: • Moved ’Instrument Controller’ microservice capability to ’Experiment Control’ • Updated ’System Manager’ and ’Data Transfer Endpoint’ capabilities • Added data storage, publishing, and transfer orchestration sequence examples in Appendix
0.8	06/30/2023	Internal draft release with the following changes: • Improved navigation to specific microservice capability definitions • Introduced ’Asynchronous Status’ interaction pattern • Updated ’Compute Allocation’ capability • Added orchestration sequence examples in Appendix
0.7	03/31/2023	Internal draft release with the following changes: • New ’Instrument Adapter’ microservice capabilities: Instrument Controller • Merge classification and catalog sections • Rename capability classes for alignment with SoS logical view • Update ’Potential Classes of INTERSECT Microservices’ figure
0.6	12/31/2022	Internal draft release with the following changes: • New ’Data and Information Management’ microservice capabilities: Data Stream, Data Transfer Orchestration, Data Transfer Endpoint • Miscellaneous fixes for grammar and spelling • Placeholder for Infrastructure Service Interaction Sequence Diagrams as Appendix
0.5	08/30/2022	Public release with the following changes: • Added figure to describe INTERSECT data model • Added figures to describe INTERSECT system hierarchy and associated entity-relationship model • New ’General Utility’ microservice capabilities: Availabilty Status, Controller Status • New ’Computing’ microservice capabilities: Container Execution, Host Command Execution • New ’Data and Information Management’ microservice capabilities: Data Catalog, Data Storage • New ’System Management’ microservice capabilities: System Manager, System Information Catalog

Version	Date	Description
0.4	06/30/2022	Internal draft release with the following changes: • Added figure to describe microservice capability definition format • Added table of generic types and structures used in capability definitions • New 'General Utility' microservice capabilities: Parameter Description, Service Configuration, UUID Generation • New 'Computing' microservice capabilities: Application Execution • New 'Data and Information Management' microservice capabilities: Entity-Relationship Catalog • New 'System Management' microservice capabilities: Availabilty Status, System Registrar • Updates to common microservice interaction patterns
0.3	04/08/2022	Internal draft release with the following changes: • Added terminology and concepts • Identified common microservice interaction patterns and message contents • Provided implementation strategies for interaction patterns using REST and PubSub communication models • Reorganized microservice classification approach to use capabilities • Refactored infrastructure computing capabilities
0.2	01/14/2022	Internal draft release
0.1	12/31/2021	Initial, unpublished draft

1 INTRODUCTION

The U. S. Department of Energy (DOE)'s artificial intelligence (AI) for Science report [31] outlines the need for intelligent systems, instruments, and facilities to enable science breakthroughs with autonomous experiments, "self-driving" laboratories, smart manufacturing, and AI-driven design, discovery and evaluation. The DOE's Computational Facilities Research Workshop report [22] identifies intelligent systems/facilities as a challenge with enabling automation and eliminating human-in-the-loop needs as a cross-cutting theme.

Autonomous experiments, "self-driving" laboratories and smart manufacturing employ machine-in-the-loop intelligence for decision-making. Human-in-the-loop needs are reduced by an autonomous online control that collects experiment data, analyzes it, and takes appropriate operational actions to steer an ongoing or plan a next experiment. It may be assisted by an AI that is trained online and/or offline with archived data and/or with synthetic data created by a digital twin. Analysis and decision making may also rely on rule-based approaches, causal models, and advanced statistical methods. Human interaction for experiment planning, observation and steering is performed through a human-machine interface.

A federated hardware/software architecture for connecting instruments with edge and center computing resources is needed that autonomously collects, transfers, stores, processes, curates, and archives scientific data in common formats. It must be able to communicate with scientific instruments and computing and data resources for orchestration and control across administrative domains, and with humans for critical decisions and feedback. Standardized communication and programming interfaces are needed that leverage community and custom software for scientific instruments, automation, workflows and data transfer. Pluggability is required to permit quickly adaptable and deployable solutions, reuse of partial solutions for different use cases, and the use of digital twins, such as a virtual instrument, robot or experiment. This federated hardware/software architecture needs to be an open standard to enable adoption by DOE's science facilities.

Oak Ridge National Laboratory (ORNL)'s INTERSECT architecture project, titled "An Open Federated Architecture for the Laboratory of the Future", creates an open federated hardware/software architecture for the laboratory of the future using a novel system of systems (SoS) and microservice architecture approach, connecting scientific instruments, robot-controlled laboratories and edge/center computing/data resources to enable autonomous experiments, "self-driving" laboratories, smart manufacturing, and AI-driven design, discovery and evaluation.

The project describes science use cases as design patterns that identify and abstract the involved hardware/software components and their interactions in terms of control, work and data flow. It creates a SoS architecture of the federated hardware/software ecosystem that clarifies terms, architectural elements, the interactions between them and compliance. It further designs a federated microservice architecture, mapping science use case design patterns to the SoS architecture with loosely coupled microservices, standardized interfaces and multi programming language support. The primary deliverable of this project is an INTERSECT Open Architecture Specification, containing the science use case design pattern catalog, the federated SoS architecture specification and the microservice architecture specification. This document represents the microservice architecture specification of the INTERSECT Open Architecture Specification.

2 INTERSECT MICROSERVICE ARCHITECTURE

2.1 INTRODUCTION TO MICROSERVICES ARCHITECTURE

Microservices architecture is a design methodology for structuring a distributed application as a networked collection of loosely-coupled services that are independently developed, maintained, and operated. Each *microservice* provides a well-defined set of functions that is domain-scoped to ensure separation of concerns between differing microservices, avoid duplicate functionality, and encourage reuse. The supported functions are defined by the *microservice contract*, which describes the purpose for each service function and associated data (e.g., request parameters and response types). A microservice may have several different implementations, where each implementation provides the same contract but uses different underlying technologies or supports a particular deployment environment. Where multiple implementations exist, an application can choose the implementation most suitable for its environment or application needs. A microservice may be self-contained such that all its functions can be serviced directly, or it may have external dependencies on other microservices that are needed to service one or more functions.

Microservices architecture is not prescriptive, but rather offers design guidelines and patterns for decomposing monolithic applications or systems into independent services, and for constructing new applications or systems through composition of independent services. A common and recommended approach for decomposition is through domain-driven design. In domain-driven design the *domain* corresponds to a particular subject area, and a subject matter expert (SME) helps to define the activities relevant to the domain, known as the domain's bounded context. An abstract conceptual model of the domain is then constructed that captures the important technical aspects of interactions with the domain. The domain model is also used to establish shared context when composing complex systems from different domains. In microservices architecture, the domain model forms the basis for defining the functions comprising the microservice contract.

Microservices are currently in wide use for cloud computing applications. There are two predominant communication architectures used for interactions between microservices: *client-server* and *asynchronous messages*. In client-server microservice communication, each interaction uses a synchronous *request-response* sequence, where one microservice (the client) issues a request of a specific type to another microservice (the server), who then handles the request and sends a specific response type based on the results of handling the request. Typical response types include formatted data that was requested (e.g., user account details or service status) or error codes. In microservice communication based on asynchronous messages, a microservice (the producer) publishes messages to a specific topic (or channel) and other microservices (the consumers) subscribe to the topic in order to register as interested parties for messages on that topic. Messages may represent service requests (e.g., control commands or information queries) or describe the occurrence of an event (e.g., a service action or state change). Due to the latter, asynchronous messaging is also commonly referred to as an event-based or event-driven communication architecture. This *publish-subscribe* mechanism enables one-to-many communications without the producer having to individually send messages to each consumer. Typically, a separate entity known as the *message broker* tracks the set of topics and the consumers subscribed to each topic, and is responsible for ensuring the delivery of topic messages to subscribers. Publish-subscribe also supports asynchronous request-response communication by using separate topics for requests and responses.

In response to these two common microservice communication architectures, two community-driven specification standards have evolved to document a microservice's contract. Both standards intend to

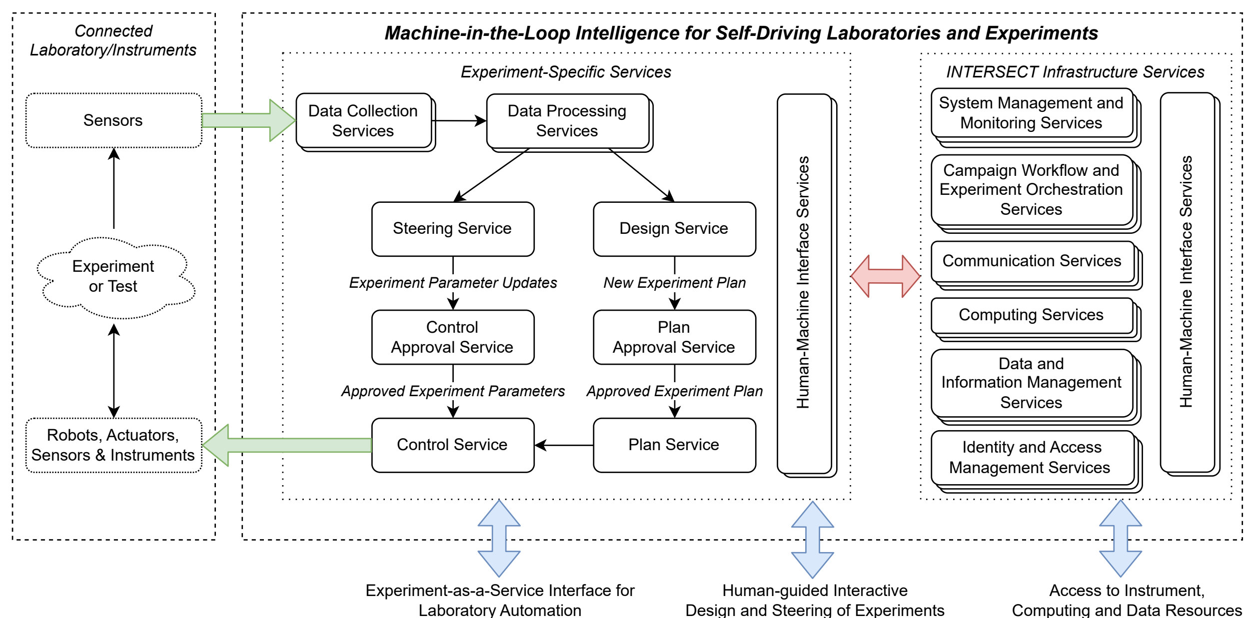

Figure 2-1. Potential Classes of INTERSECT Microservices

provide a programming language-agnostic method for application programming interface (API) definition that is understandable to both humans and machines, which enables a wide variety of automated tooling for code development, testing, and documentation. The OpenAPI Specification [30] targets client-server microservices that provide a Representational State Transfer (REST) API accesible via Hypertext Transfer Protocol (HTTP). The AsyncAPI Specification [28] targets microservices based on asynchronous messaging and supports a wide variety of messaging protocols. AsyncAPI was originally derived from OpenAPI and thus shares much terminology. In some cases, AsyncAPI directly supports referencing components of OpenAPI specifications such as data schemas.

2.2 MICROSERVICES ARCHITECTURE IN INTERSECT

Within the INTERSECT Open Architecture, the microservice architecture specification provides a catalog of infrastructure and experiment-specific microservices that may be useful within an interconnected science ecosystem. All microservices are defined to facilitate composition within federated SoS architectures, where each subsystem corresponds to one or more coordinating microservices. INTERSECT infrastructure microservices represent common service functionality and capabilities, such as data management, computing, system management, and workflow orchestration that are likely to be generally useful across many science ecosystems without the need for customization. Experiment-specific microservices, on the other hand, represent services whose implementation may require detailed application knowledge, such as experiment planning or steering services that require knowledge of experiment-specific control parameters and their associated constraints. The INTERSECT science use case design patterns help identify the relevant infrastructure and experiment-specific microservices for a given science ecosystem. Figure 2-1 provides an architectural overview of the potential classes of microservices that may be involved with a given interconnected science ecosystem. Section 3 provides additional information on the various INTERSECT microservice classes and their shared capabilities and data requirements.

2.3 COMMONALITIES OF INTERSECT MICROSERVICES

To enable federation of INTERSECT microservices, it is useful to understand the types of interactions a given microservice may reasonably expect from one of its clients. As shown in Figure 2-2, we have identified three common patterns that substantively cover the expected interactions: *Command*, *Request-Reply*, and *Asynchronous Status or Event*. The *Command Interaction Pattern* involves the client asking the microservice to do something. The microservice typically responds immediately with a simple acknowledgement that the command has been received successfully or some error status indicating why the command was not acceptable. A command may initiate an activity within the microservice, but that activity is not ordered with respect to the command acknowledgment message. Thus, commands are asynchronous interactions from the client perspective. The *Request-Reply Interaction Pattern* has the client making a request of the microservice that includes an expected reply containing pertinent information or data related to the request. Because the reply is not sent until the request has been fully processed, this is a synchronous interaction pattern from the client perspective. Finally, the *Asynchronous Status or Event Interaction Pattern* represents cases where the microservice generates status or event information that is broadcast to any interested parties at irregular intervals as a result of internal operational state changes or ongoing activities. Events are informational in nature and there is no expectation that the associated message must be delivered. However, status messages are typically associated with activities initiated by clients, and therefore must provide some limited form of message durability to ensure that the message is delivered to at least one interested party. Each of these interaction patterns supports implementations based on RESTful client-server communication or asynchronous messaging, as described later in Section 4.1.

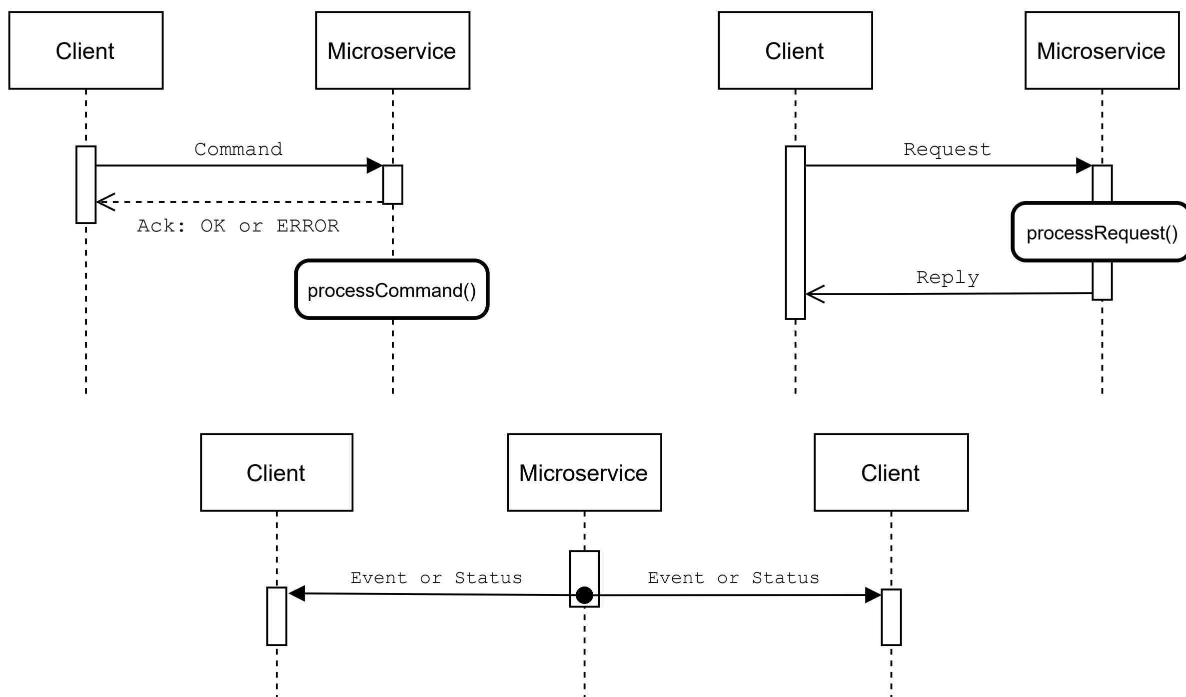

Figure 2-2. Interaction Patterns for INTERSECT Microservices

The messages used in these interaction patterns also share common information that should be included in the message contents. For instance, in all three patterns, messages should include information that

describes the source of the message and the specific type of `Command`, `Request`, `Reply`, `Event`, or `Status`. For a `Command` message, the type should indicate the requested action. For a `Request` or `Reply` message, the type should indicate the requested information or data. For an `Event` message, the type should identify the generator of the event. For a `Status` message, the type should identify the activity or resource whose status is being reported. This type information can also be used by the receiver to know what other information may be included in the message, such as any parameters or data associated with a `Command`, `Request` or `Reply`, or codes and descriptions associated with a particular `Event`. Similarly, a timestamp associated with the sending of the initial message is useful in all three patterns. For a `Command` or `Request` message, this timestamp represents the time when a client issued the request, which may be used in situations requiring a completion deadline or for communication retry purposes. For an `Event` or `Status` message, the timestamp indicates when the event or status change occurred. Finally, it is useful within *Command* and *Request-Reply* interactions to support trace identifiers that clients can use to associate messages with specific client state. Trace identifiers are particularly useful when a client interaction requires the target microservice to make further requests of other microservices. The target microservice should pass on the trace identifier it received from the client along with its own external requests. When combined with a distributed message logging facility, the trace identifier enables a complete view of the request progress from the initial client, through one or more remote microservice requests and replies, and ending with a reply to the initial client. This complete traceability of requests through microservices is also crucial for operational insight when investigating and resolving problems encountered in deployed microservices architectures.

3 CLASSIFICATION OF INTERSECT MICROSERVICES

We first describe the format used to document microservice capabilities in Section 3.1. In the following two sections that focus on microservice capabilities for infrastructure services (Section 3.2) and experiment-specific services (Section 3.3), we classify capabilities into groups based on their general purpose, such as computation or data management for infrastructure services, and experimental control or design for experiment-specific services.

3.1 INTERSECT MICROSERVICE CAPABILITY DEFINITION FORMAT

As previously described in Section 2.1, a *Microservice* provides a well-defined set of functionality that is scoped to focus on a particular domain of interest. The set of *Microservice Functions* with associated input and output data is known as the *Microservice Contract*. In the following subsections, we define microservice capabilities that describe the purpose and proposed functionality for a wide range of unique focus domains. For each capability, the proposed functionality (i.e., methods and associated data) is grouped by the corresponding microservice interaction pattern. Together, the functionality definitions can serve as the basis for an implementation's microservice contract. Where applicable, we also indicate relationships to other microservice capabilities, such as functionality extensions or required dependencies.

The functionality provided by a microservice will necessarily change over time to adapt to new requirements or use cases. As a result, each microservice capability definition should be versioned to ensure that providers and clients of the capability can agree on a consistent set of the included functionality and any custom data type defintions. A suitable approach to versioning of microservice APIs is that defined by Semantic Versioning [1].

Figure 3-1 is an example of the capability definition format that describes the information provided by each element of the definition. The data types used in defining each capability are generic names for common types and structures supported by the data models of most data schema standards (e.g., JSON Schema [6], XML Schema Definition (XSD) [33], and Apache Avro [7]). To avoid confusion with specific schema data model types, Table 3-1 provides a description for each generic type or structure and lists compatible specific types for a few data schema formats commonly used in microservice messaging.

Capability: ***Capability Name***

Description: A short summary description of the domain of interest for this capability and the provided functionality.

Version: A meaningful version number for the capability and its provided interaction methods and data types. For new versions, the changes from the previous version (i.e., additions, deprecations, or deletions) should be enumerated.

Related Capabilities: Where applicable, provides references to related capabilities.

- *Extends*: A list of base capabilities that the functionality of this capability extends. A service implementing this capability must also implement the base capabilities.
- *Requires*: A list of required capabilities that are necessary to implement the functionality of this capability. The required capabilities are most often provided by other services, but may be implemented in the same service. Where necessary, specific versions of the required capabilities may be specified.

Custom Data Type: Where applicable, provides definitions of new data types or structures.

Interactions: Command

- `MethodName()`

  Purpose: A short description of the purpose of the current command method.

  Command Data: A list of input data for the current method formatted as:

  - `dataName` (`DataType`) : A description of the data, including any format or value constraints.

Interactions: Request-Reply

- `MethodName()`

  Purpose: A short description of the purpose of the current request method.

  Request Data: A list of input data for the current method formatted as:

  - `dataName` (`DataType`) : A description of the data, including any format or value constraints.

  Reply Data: A list of output data for the current method formatted as:

  - `dataName` (`DataType`) : A description of the data, including any format or value constraints.

Interactions: Asynchronous Event

- `EventName`

  Purpose: A description of the event and its cause, if any.

  Event Data: A list of pertinent data for the current event formatted as:

  - `dataName` (`DataType`) : A description of the data, including any format or value constraints.

Interactions: Asynchronous Status

- `StatusName`

  Purpose: A description of the activity status or state change.

  Status Data: A list of pertinent data for the current status formatted as:

  - `dataName` (`DataType`) : A description of the data, including any format or value constraints.

Figure 3-1. Microservice Capability Definition Format

Data Type or Structure	Description	JSON Schema Types	XSD Types	Apache Avro Types
Boolean	A logical value representing True or False values.	boolean	boolean	boolean
Bytes	An ordered sequence of byte values.	string (Base64 encoded)	hexBinary, base64Binary	bytes
Integer	An integral numeric value, preferably supporting up to 64-bit values.	number	decimal	long
Filepath	An absolute path to a local file system entry (i.e., a file or directory).	string	string	string
Float	A floating point numeric value, preferably supporting double precision.	number	double	double
KeyVal<type>	A uniquely-named String key paired with an associated value of the given type.	object: {key, value}	<key> element	map
List<type>	A list of items of the given type.	array	list, sequence	array
String	A character string of unspecified length, preferably encoded using Unicode.	string	string	string
TimeDuration	A value representing the difference between a start and end TimeStamp, preferably with a minimum resolution of seconds.	string (format: duration)	duration	string (logicalType: duration)
TimeStamp	A timestamp value, preferably in Universal Coordinated Time (UTC) format.	string (format: date-time)	dateTimeStamp	string (logicalType: timestamp-millis)
UUID	A universally unique identifier (RFC 4122).	string (format: uuid)	string	string (logicalType: uuid)

Table 3-1. Generic Data Types and Structures and Associated Schema-specific Representations

3.2 INTERSECT INFRASTRUCTURE MICROSERVICE CAPABILITIES

In this section, we classify INTERSECT infrastructure microservices into five groups according to their purpose: (1) General Utility, (2) INTERSECT Data and Information Management, (3) INTERSECT System Management, and (4) INTERSECT Resource Adapters.

3.2.1 General Utility

General utility microservice capabilities represent functionality that may be useful to support a wide range of infrastructure or experiment-specific services. They are not intended to exist in isolated services, rather other capabilities should include their functionality through an *Extends* relationship.

3.2.1.1 Capability: *Availability Status*

Description: Provides inspection and control of the availability status of a service. **NOTE:** All services are expected to provide this capability.

Custom Data Type: `AvailabilityStatus`

- `uuid` (`UUID`) : The UUID of the entity reporting its status.
- `currentStatus` (`String`) : The current status.
- `previousStatus` (`String`) : The previous status.
- `statusDescription` (`String`) : Additional information related to the current status (e.g., why a status change occurred).

Interactions: Command

- `SetAvailabilityStatus()`

  Purpose: Update the status using the given `statusCode`. On success, triggers the `AvailabilityStatusChange` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `status` (`String`) : The new status (e.g., "AVAILABILITY_UNKNOWN", "AVAILABLE", "PARTIAL_AVAILABILITY", or "UNAVAILABLE").
  - `statusDescription` (`String`) : (Optional) A description of the reason for the status change.

Interactions: Request-Reply

- `GetAvailabilityStatus()`

  Purpose: Request the current availability status.

  Reply Data: `dataName` (Data Type) : Data description.

  - `status` (`AvailabilityStatus`) : The current availability status information.

Interactions: Asynchronous Status

- `AvailabilityStatusChange`

  Purpose: Notification of changes to availability status.

  Event Data: `dataName` (Data Type) : Data description.

  - `status` (`AvailabilityStatus`) : The current availability status information.

3.2.1.2 Capability: *Controller Status*

Description: Provides inspection and control of the controller status of a system or service. **NOTE:** All services are expected to provide this capability.

Custom Data Type: `ControllerStatus`

- `uuid` (`UUID`) : The UUID of the entity reporting its status.
- `primaryController` (`UUID`) : The UUID of the primary controller.
- `secondaryController` (`UUID`) : The UUID of the secondary controller.
- `statusDescription` (`String`) : Additional information related to the current controller change event.

Interactions: Command

- `SetPrimaryController()`

  Purpose: Change the primary controller to the given `primaryUUID`. On success, triggers the `ControllerStatusChange` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `primaryUUID` (`UUID`) : The UUID of the new primary controller.
  - `controllerSecret` (`Bytes`) : A secret for use in validating primary control changes.
  - `description` (`String`) : (Optional) Additional information for use in generating the ControllerStatusChange event.

- `ReleaseSecondaryControl()`

  Purpose: Release secondary control permission for the given `secondaryUUID`. On success, triggers the `ControllerStatusChange` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `secondaryUUID` (`UUID`) : The UUID of the new secondary controller.
  - `controllerSecret` (`Bytes`) : A secret for use in validating secondary control changes.
  - `description` (`String`) : (Optional) Additional information for use in generating the ControllerStatusChange event.

Interactions: Request-Reply

- `AcquireSecondaryControl()`

  Purpose: Request that the given `secondaryUUID` be given secondary control permission. On success, triggers the `ControllerStatusChange` event.

  Request Data: `dataName` (Data Type) : Data description.

- `secondaryUUID` (`UUID`) : The UUID of the new secondary controller.
- `controllerSecret` (`Bytes`) : A secret for use in validating secondary control changes.
- `description` (`String`) : (Optional) Additional information for use in generating the ControllerStatusChange event.

Reply Data: `dataName` (Data Type) : Data description.

- `errorMsg` (`String`) : (Optional) An error message describing why secondary control permission could not be granted.

- `GetControllerStatus()`

  Purpose: Request the current controller status.

  Reply Data: `dataName` (Data Type) : Data description.

  - `status` (`ControllerStatus`) : The current controller status information.

Interactions: Asynchronous Status

- `ControllerStatusChange`

  Purpose: Notification of changes to controller status.

  Event Data: `dataName` (Data Type) : Data description.

  - `status` (`ControllerStatus`) : The current controller status information.

3.2.1.3 Capability: *Parameter Configuration*

Description: Provides description and configuration of service parameters, allowing clients to query the supported parameters and their associated constraints and to set their values. Service parameters represent settings that affect the behavior of the service independent of any of its underlying capability interaction methods. Service parameters are defined in a generic fashion to allow representation of a wide variety of uses, including but not limited to static configuration settings, dynamic or runtime settings, instrument control parameters, machine learning model parameters, etc.

Custom Data Type: `ParameterInfo`

- `paramName` (`String`) : The name of the parameter. The name must be unique among all parameters supported by a given service (which may implement several capabilities).
- `paramDescription` (`String`) : A detailed description of the parameter's intended usage, including a discussion of its type and acceptable values.
- `paramType` (`String`) : The name of the generic data type for the parameter.
- `defaultValue` (`String`) : A string representation of the parameter's default value.
- `permittedValues` (`List<String>`) : An optional list of strings that enumerates acceptable values for the parameter. For numeric parameters, each string may represent a value range (e.g., "[0,100]").

Interactions: Command

- `AddConfigurationParameters()`

  Purpose: Add the given parameters to the service's supported configuration set.

  Command Data: `dataName` (Data Type) : Data description.

  - `parameters` (`List<ParameterInfo>`) : A list of parameter information structures.

- `SetConfigurationValues()`

  Purpose: Update the service's configuration using the given parameter name-value list.

  Command Data: `dataName` (Data Type) : Data description.

  - `configValues` (`List< KeyVal<String> >`) : The list of configuraton parameter name-value `String` pairs.

Interactions: Request-Reply

- `GetParameterNames()`

  Purpose: Request a list of the service's supported configuration parameter names.

  Reply Data: `dataName` (Data Type) : Data description.

  - `paramNames` (`List<String>`) : The list of supported parameter names.

- `GetParameterDetails()`

Purpose: Request detailed information for the given list of `paramNames`.

Request Data: `dataName` (Data Type) : Data description.

- `paramNames` (`List<String>`) : A list of service parameter names.

Reply Data: `dataName` (Data Type) : Data description.

- `paramDetails` (`List<ParameterInfo>`) : A list of parameter information structures for the requested service parameters.

- `GetConfiguration()`

  Purpose: Request the current values for all service configuration parameters.

  Reply Data: `dataName` (Data Type) : Data description.

  - `configValues` (`List< KeyVal<String> >`) : A list of name-value `String` pairs for all supported service parameters.

- `GetConfigurationValues()`

  Purpose: Request the current values for the given list of `configNames`.

  Request Data: `dataName` (Data Type) : Data description.

  - `configNames` (`List<String>`) : A list of service configuration parameter names.

  Reply Data: `dataName` (Data Type) : Data description.

  - `configValues` (`List< KeyVal<String> >`) : A list of name-value `String` pairs for the requested service parameters.

- `ValidateParameters()`

  Purpose: Validate the given list of service configuration parameter name-value pairs.

  Request Data: `dataName` (Data Type) : Data description.

  - `paramValues` (`List< KeyVal<String> >`) : A list of parameter name-value pairs.

  Reply Data: `dataName` (Data Type) : Data description.

  - `invalidCount` (`Integer`) : The number of supplied parameters that were invalid.
  - `invalidParams` (`List< KeyVal<String> >`) : A list of name-reason pairs for the supplied parameters that were determined to be invalid.

3.2.1.4 Capability: *UUID Generation*

Description: Provides generation of universally unique identifier (UUID) according to RFC 4122 [23].

Interactions: Request-Reply

- `GetNamespaceUUID()`

  Purpose: Request a new name-based (i.e., version 3 or version 5) UUID value.

  Request Data: `dataName` (Data Type) : Data description.

  - `namespaceId` (`UUID`) : The namespace base UUID base.
  - `name` (`String`) : The unique name within the namespace to use in generation of the UUID.

  Reply Data: `dataName` (Data Type) : Data description.

  - `id` (`UUID`) : The new UUID.
  - `errorMsg` (`String`) : An error message describing why UUID generation failed.

- `GetUUID()`

  Purpose: Request a new time-based (i.e., version 1) UUID value.

  Reply Data: `dataName` (Data Type) : Data description.

  - `id` (`UUID`) : The new UUID.
  - `errorMsg` (`String`) : An error message describing why UUID generation failed.

3.2.2 INTERSECT Data and Information Management

Data and information management services support the storage, transfer, and introspection needs of data, metadata, and information produced or consumed within INTERSECT campaign ecosystems.

3.2.2.1 Data Storage Services

Data storage services generally store and retrieve data items or data collections in one or more data namespaces. The services may optionally provide data compression for efficient use of storage space, data-at-rest encryption for secure storage, and data versioning. Three common types of data storage services include file storage, key-value storage, and object storage.

File storage services store and retrieve files in a hierarchical file system namespace. File names are strings that are unique only within their enclosing directory. Related technologies include archival storage systems (e.g., High Performance Storage System (HPSS)), local file systems, cloud file storage and sharing services (e.g., DropBox [18], Google Drive [12], and Microsoft OneDrive [26]), and distributed or parallel file systems (e.g., General Parallel File System (GPFS), Lustre, and Network File System (NFS)).

Key-value storage services store and retrieve key-value pairs in a flat namespace. The unique keys are often strings, but may be arbitrary data blobs. Values may be arbitrary data blobs or have service-specific structure. These services typically provide volatile memory-based storage, but some offer data durability functionality to store data on stable storage devices such as hard disk drive (HDD) or non-volatile memory (NVM) devices. Related technologies include popular cloud infrastructures such as Memcached [5] and Redis [24].

Object storage services store and retrieve named objects or object collections in a flat namespace. Object names are unique strings within their namespace. The data associated with objects may be arbitrary blobs, including structured and unstructured data. Relevant technologies include Amazon S3 [17], Google Cloud Storage [11], Microsoft Azure Blob Storage [25], and MinIO [19].

3.2.2.2 Data Transfer Services

Data transfer services facilitate the transfer of data products or data streams from producers to consumers. The services may optionally provide data compression for efficient use of network resources and encryption for secure data communication. Two common types of data transfer services include file transfer and data streaming.

File transfer services are designed to transfer files from one file storage service to another file storage service, typically via a custom file transfer protocol such as File Transfer Protocol (FTP) or GridFTP [2]. These services may support batched file transfers containing multiple files or entire directory hierarchies. Relevant technologies include Globus [32] and file transfers based on HTTP, Hypertext Transfer Protocol Secure (HTTPS), and Secure Sockets Layer (SSL).

Data streaming services provide streaming data transfers from one source service to another destination service. The data streams may be raw, object- or structure-based, or file-based. Some services support concurrent transfers of multiple data streams between a given source and destination. Relevant technologies include cloud data and event streaming frameworks such as Apache Kafka [9] and Amazon Kinesis [15].

3.2.2.3 Database and Information Management Services

Database and information management services store, retrieve, and support queries over data and associated metadata with potentially complex relationships. These services may optionally provide data indexing or compute-in-data features to optimize query performance. Relevant technologies include relational databases (e.g., MySQL [27], PostgreSQL [13], and SQLite [3]) and non-relational document stores (e.g., MongoDB [20]), columnar stores (e.g., Amazon DynamoDB [14], Apache Cassandra [8], Google BigTable [10]), and graph databases (e.g., Neo4J [21] and Amazon Neptune [16]).

3.2.2.4 A Data Model for INTERSECT Microservices

The INTERSECT microservice capabilities for data management are designed around a flexible, generalized data model as shown in Figure 3-2. A campaign workflow produces or consumes data from one or more *Data Namespaces*, which are uniquely-named logical containers for *Data Items*, *Data Collections*, and *Data Streams*. *Data Items* represent uniquely-named pieces of data such as files, objects, or key-value pairs. *Data Collections* are uniquely-named collections of *Data Items*, and correspond to grouping mechanisms such as directories, object buckets, or table records. *Data Streams* represent

uniquely-named streaming data sources that can support multiple stream consumers. The data stream elements are *Data Items* with names corresponding to the element indices.

Data Namespaces are uniquely named logical containers for *Data Items*, *Data Collections*, and *Data Streams*.

Data Items represent uniquely named pieces of data such as files, objects, or key-value pairs.

Data Collections are uniquely named groups of items, and correspond to grouping mechanisms such as directories, object buckets, or table records.

Data Streams represent uniquely named streaming data sources. The data stream elements are *Data Items* with names corresponding to the element indices.

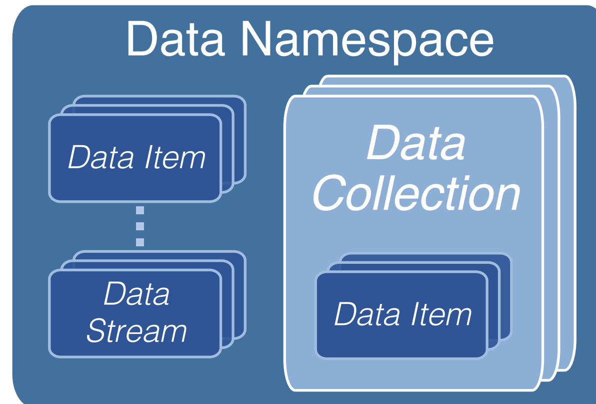

Figure 3-2. INTERSECT Data Model

The microservice capabilities for information management represent well-established methods such as Entity-Relationship (E-R) models and both relational and non-relational databases. E-R models provide a flexible representation of arbitrary information about unique entities and their relationships to other entities. Relational databases provide atomic, consistent, idempotent, and durable (ACID) transactions and complex queries over structured relations defined by a schema in one or more tables, and may support advanced data indexing for query performance optimization. Non-relational databases provide non-transactional data updates and queries over unstructured data and information such as objects, columnar data, time-series data, or graphs.

To facilitate workflows across interconnected science ecosystems, each INTERSECT system is expected to provide a *Data Management* service that supports the data storage and transfer needs of services and applications within the system. Furthermore, the *Campaign Orchestrator* service that orchestrates the execution of campaign workflows is expected to manage a catalog of *Data Products* corresponding to *Data Items* or *Data Collections* and orchestrate data movement between producers and consumers. Additional details on the expected microservice interactions related to campaign data management can be found in Appendix A-3.

3.2.2.5 Capability: *Entity-Relationship Catalog*

Description: Provides a generic E-R information catalog that maintains names, descriptions, properties, and labels for uniquely-identified entities and tracks named relationships between entities.

Custom Data Type: `CatalogEntityInformation`

- `entityName` (`String`) : The name of the entity.
- `entityType` (`String`) : The type name of the entity.
- `entityDescription` (`String`) : The description of the entity.
- `entityProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs for the entity's properties.
- `entityLabels` (`List<String>`) : (Optional) A list of `String` labels for the entity.

Custom Data Type: `CatalogEntityRelationship`

- `relationName` (`String`) : The name of the relation.
- `sourceId` (`UUID`) : The unique id for the source entity.
- `targetId` (`UUID`) : The unique id for the target entity.

Interactions: Command

- `CreateEntity()`

  Purpose: Create a new entity with the given id and information (i.e., name, type, description, properties, and labels). On successful creation, triggers the `CatalogEntityCreation` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `entityId` (`UUID`) : The unique id for the entity.
  - `entityInfo` (`CatalogEntityInformation`) : The entity information.

- `RemoveEntity()`

  Purpose: Remove the entity with given `entityId`. Also removes any relationships associated with the removed entity. On successful removal, triggers the `CatalogEntityRemoval` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `entityId` (`UUID`) : The unique id of the entity to remove.

- `CreateRelation()`

  Purpose: Create a new named relation between the given source and target entities. On successful creation, triggers the `CatalogRelationCreation` event.

  Command Data: `dataName` (Data Type) : Data description.

- relation (CatalogEntityRelationship) : The relation to add, which identifes the relation name and the source and target entity ids.

- RemoveRelation()

  Purpose: Remove the relationship(s) having the given name. If both the sourceId and targetId are provided, removes only the relationship between those entities. If only the sourceId is provided, removes all relationships with the given name that are sourced at that entity. If only the targetId is provided, removes all relationships with the given name that are targeted at that entity. Also removes any relationships associated with the removed entity. On successful removal, triggers the CatalogRelationRemoval event.

  Command Data: dataName (Data Type) : Data description.

  - relationName (String) : The name of the relation to remove.
  - sourceId (UUID) : The unique id of the relation's source entity, or the ALL_UUID value to represent all sources.
  - targetId (UUID) : The unique id of the relation's target entity, or the ALL_UUID value to represent all targets.

Interactions: Request-Reply

- GetEntityInformation()

  Purpose: Get the information of the entity with given entityId.

  Request Data: dataName (Data Type) : Data description.

  - entityId (UUID) : The unique id for the entity.

  Reply Data: dataName (Data Type) : Data description.

  - entityInfo (CatalogEntityInformation) : The entity information.

- GetEntityRelationships()

  Purpose: Get the source and target relationships of the entity with given entityId. Optionally, return only the relations with the given relationName.

  Request Data: dataName (Data Type) : Data description.

  - entityId (UUID) : The unique id for the entity.
  - relationName (String) : (Optional) The name of a specific relation for which to return results.

  Reply Data: dataName (Data Type) : Data description.

  - relationshipList (List<CatalogEntityRelationship>) : The list of entity relationships.

- GetEntitySourceRelationships()

Purpose: Get the relationships that are sourced from the entity with given `entityId`. Optionally, return only the relations with the given `relationName`.

Request Data: `dataName` (Data Type) : Data description.

- `entityId` (`UUID`) : The unique id for the entity.
- `relationName` (`String`) : (Optional) The name of a specific relation for which to return results.

Reply Data: `dataName` (Data Type) : Data description.

- `relationshipList` (`List<CatalogEntityRelationship>`) : The list of entity source relationships.

• `GetEntityTargetRelationships()`

Purpose: Get the relationships that target the entity with given `entityId`. Optionally, return only the relations with the given `relationName`.

Request Data: `dataName` (Data Type) : Data description.

- `entityId` (`UUID`) : The unique id for the entity.
- `relationName` (`String`) : (Optional) The name of a specific relation for which to return results.

Reply Data: `dataName` (Data Type) : Data description.

- `relationshipList` (`List<CatalogEntityRelationship>`) : The list of entity target relationships.

• `GetSourceEntitiesByRelation()`

Purpose: Get the entities that are sources for relationships with the given `relationName`.

Request Data: `dataName` (Data Type) : Data description.

- `relationName` (`String`) : The name of a specific relation for which to return results.

Reply Data: `dataName` (Data Type) : Data description.

- `sourceIds` (`List<UUID>`) : The list of source entities.

• `GetTargetEntitiesByRelation()`

Purpose: Get the entities that are targets for relationships with the given `relationName`.

Request Data: `dataName` (Data Type) : Data description.

- `relationName` (`String`) : The name of a specific relation for which to return results.

Reply Data: `dataName` (Data Type) : Data description.

- `targetIds` (`List<UUID>`) : The list of target entities.

• `GetEntitiesByType()`

Purpose: Get the entities with the given `entityType`.

Request Data: `dataName` (Data Type) : Data description.

- `entityType` (`String`) : The type name of the requested entities.

Reply Data: `dataName` (Data Type) : Data description.

- `entityIds` (`List<UUID>`) : The list of entities of the requested type.

- `GetEntitiesByLabel()`

Purpose: Get the entities whose labels include the given `labelName`.

Request Data: `dataName` (Data Type) : Data description.

- `labelName` (`String`) : The name of the label.

Reply Data: `dataName` (Data Type) : Data description.

- `entityIds` (`List<UUID>`) : The list of entities with the requested label.

- `GetEntitiesByProperty()`

Purpose: Get the entities whose properties include the given `propertyName`. Optionally, only return entities whose property value matches the `valueExpression`.

Request Data: `dataName` (Data Type) : Data description.

- `propertyName` (`String`) : The name of the property.
- `valueExpression` (`String`) : (Optional) An expression to evaluate against the property value.

Reply Data: `dataName` (Data Type) : Data description.

- `entityIds` (`List<UUID>`) : The list of entities with the requested property.

Interactions: Asynchronous Event

- `CatalogEntityCreation`

Purpose: Notification of catalog entity creations.

Event Data: `dataName` (Data Type) : Data description.

- `entityId` (`UUID`) : The unique id of the created entity.
- `entityName` (`String`) : The name of the created entity.
- `entityType` (`String`) : The type name of the created entity.

- `CatalogRelationCreation`

Purpose: Notification of catalog relationship creations.

Event Data: `dataName` (Data Type) : Data description.

- `relation` (`CatalogEntityRelationship`) : The relation that was created.

- `CatalogEntityRemoval`

  Purpose: Notification of catalog entity removals.

  Event Data: `dataName` (Data Type) : Data description.

  - `entityId` (`UUID`) : The unique id of the removed entity.
  - `entityName` (`String`) : The name of the removed entity.
  - `entityType` (`String`) : The type name of the removed entity.

- `CatalogRelationRemoval`

  Purpose: Notification of catalog relationship removals.

  Event Data: `dataName` (Data Type) : Data description.

  - `relationName` (`String`) : The name of the removed relation.
  - `sourceId` (`UUID`) : The unique id of the relation's source entity, or the `ALL_UUID` value to represent all sources.
  - `targetId` (`UUID`) : The unique id of the relation's target entity, or the `ALL_UUID` value to represent all targets.

3.2.2.6 Capability: *Data Catalog*

Description: Provides information on published data products and data streams.

Related Capabilities: Requires

- *Entity-Relationship Catalog*

Interactions: Command

- `PublishDataProduct()`

  Purpose: Publish the availability of a data product (i.e., a data item or data collection). Successful publication will trigger the `DataCatalogProductAvailable` event and creation of a `DataItem` or `DataItemCollection` entity with the given information and a relationship to the new entity with the parent `DataNamespace`.

  Command Data: `dataName` (Data Type) : Data description.

  - `productDescription` (`String`) : A user-friendly description of the product.
  - `productId` (`UUID`) : The UUID of the data product.
  - `serviceId` (`UUID`) : The UUID of the Data Storage service hosting the data product.
  - `namespaceName` (`String`) : The name of the data namespace containing the product.
  - `collectionName` (`String`) : (Optional) The name of the data collection for the product.
  - `itemName` (`String`) : (Optional) The name of the data item for the product.
  - `productLabels` (`List<String>`) : (Optional) A list of `String` labels for the data product.
  - `productProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs for the data product's properties.

- `PublishDataStream()`

  Purpose: Publish the availability of a data stream. Successful publication will trigger the `DataCatalogStreamAvailable` event and creation of a `DataStream` entity with the given information and a relationship to the new entity with the parent `DataNamespace`.

  Command Data: `dataName` (Data Type) : Data description.

  - `streamDescription` (`String`) : A user-friendly description of the data stream.
  - `streamId` (`UUID`) : The UUID of the data product.
  - `serviceId` (`UUID`) : The UUID of the Data Stream service hosting the data stream.
  - `namespaceName` (`String`) : The name of the data namespace containing the stream.
  - `streamName` (`String`) : The name of the data stream.

- streamLabels (List<String>) : (Optional) A list of String labels for the data stream.
- streamProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs for the data streams's properties.

- RemoveDataProduct()

  Purpose: Remove a data product from the catalog. Successful removal will trigger the DataCatalogProductRemoval event.

  Command Data: dataName (Data Type) : Data description.

  - productId (UUID) : The UUID of the data product.

- RemoveDataStream()

  Purpose: Remove a data product from the catalog. Successful removal will trigger the DataCatalogStreamRemoval event.

  Command Data: dataName (Data Type) : Data description.

  - streamId (UUID) : The UUID of the data stream.

Interactions: Request-Reply

- CreateDataProductUUID()

  Purpose: Create a UUID for a new data product (i.e., a data item or data collection).

  Reply Data: dataName (Data Type) : Data description.

  - productId (UUID) : The assigned UUID for the data product.
  - errorMsg (String) : An error message describing why UUID creation failed.

- CreateDataStreamUUID()

  Purpose: Create a UUID for a new data stream.

  Reply Data: dataName (Data Type) : Data description.

  - streamId (UUID) : The assigned UUID for the data stream.
  - errorMsg (String) : An error message describing why UUID creation failed.

- GetDataProductInformation()

  Purpose: Get the full set of data catalog information for the given productId.

  Request Data: dataName (Data Type) : Data description.

  - productId (UUID) : The UUID of the data product.

  Reply Data: dataName (Data Type) : Data description.

  - productInfo (CatalogEntityInformation) : The catalog information for the data product.

- errorMsg (String) : An error message describing why the lookup failed.

- GetDataStreamInformation()

  Purpose: Get the full set of data catalog information for the given streamId.

  Request Data: dataName (Data Type) : Data description.

  - streamId (UUID) : The UUID of the data stream.

  Reply Data: dataName (Data Type) : Data description.

  - streamInfo (CatalogEntityInformation) : The catalog information for the data stream.
  - errorMsg (String) : An error message describing why the lookup failed.

Interactions: Asynchronous Status

- DataCatalogProductAvailable

  Purpose: Notification of published data products.

  Event Data: dataName (Data Type) : Data description.

  - productId (UUID) : The UUID of the data product.
  - serviceId (UUID) : The UUID of the Data Storage service hosting the data product.
  - namespaceName (String) : The name of the data namespace containing the product.
  - collectionName (String) : (Optional) The name of the data collection for the product.
  - itemName (String) : (Optional) The name of the data item for the product.

- DataCatalogProductRemoval

  Purpose: Notification of data product removal.

  Event Data: dataName (Data Type) : Data description.

  - productId (UUID) : The UUID of the data product.

- DataCatalogStreamAvailable

  Purpose: Notification of published data streams.

  Event Data: dataName (Data Type) : Data description.

  - streamId (UUID) : The UUID of the data stream.
  - serviceId (UUID) : The UUID of the Data Stream service hosting the data stream.
  - namespaceName (String) : The name of the data namespace containing the stream.
  - streamName (String) : The name of the data stream.

- DataCatalogStreamRemoval

Purpose: Notification of data stream removal.

Event Data: `dataName` (Data Type) : Data description.

- `streamId` (`UUID`) : The UUID of the data stream.

3.2.2.7 Capability: *Data Storage*

Description: Provides storage for data items and collections of data items within one or more namespaces as shown in Figure 3-2.

Custom Data Type: DataCollectionDetails

- collectionName (String) : The name of the data item collection.
- collectionNamespace (String) : The name of the namespace that contains the collection.
- collectionItemCount (Integer) : The number of data items in the collection.
- collectionSize (Integer) : The aggregate size in bytes of all the data items in the collection.
- collectionCreationTime (TimeStamp) : Creation timestamp for the collection.
- collectionProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs for the collection's properties.

Custom Data Type: DataItemDetails

- itemName (String) : The name of the data item.
- itemNamespace (String) : The name of the namespace that contains the data item.
- itemSize (Integer) : The size in bytes of the data item.
- itemCreationTime (TimeStamp) : Timestamp of the data item's creation.
- itemUpdateTime (TimeStamp) : (Optional) Timestamp of the most recent update data item.
- itemCollection (String) : (Optional) When part of a collection, the name of the collection that contains the data item.
- itemProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs for the data item's properties.

Custom Data Type: DataNamespaceDetails

- namespaceName (String) : The name of the data item collection.
- namespaceCollectionCount (Integer) : The number of data item collections in the namespace.
- namespaceItemCount (Integer) : The total number of data items in the namespace, including all collections.
- namespaceSize (Integer) : The aggregate size in bytes of all the data items in the namespace, including all collections.

Interactions: Command

- CreateDataCollection()

Purpose: Create a new data item collection in the given namespace. On successful creation, triggers the `DataCollectionCreation` event.

Command Data: `dataName` (Data Type) : Data description.

- `collectionName` (`String`) : The name of the data item collection.
- `collectionNamespace` (`String`) : The name of the namespace in which to place the collection.
- `collectionProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs to use as the collection's properties.

- `CreateDataItemFromBytes()`

Purpose: Create a new data item with provided contents. The item is placed in the given namespace and optional collection. On successful creation, triggers the `DataItemCreation` event. (NOTE: An implementation may limit the maximum size of a data item that can be created from bytes.)

Command Data: `dataName` (Data Type) : Data description.

- `itemName` (`String`) : The name of the data item.
- `itemNamespace` (`String`) : The name of the namespace in which to place the data item.
- `contentBytes` (`Bytes`) : The data item contents.
- `itemCollection` (`String`) : (Optional) The name of the collection in which to place the data item.
- `itemProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs to use as the data item's properties.

- `CreateDataItemFromLocalFile()`

Purpose: Create a new data item associated with an existing local file. The item is placed in the given namespace and optional collection. On successful creation, triggers the `DataItemCreation` event.

Command Data: `dataName` (Data Type) : Data description.

- `itemName` (`String`) : The name of the data item.
- `itemNamespace` (`String`) : The name of the namespace in which to place the data item.
- `localFilePath` (`Filepath`) : The absolute path of the local file from which to create the data item.
- `itemCollection` (`String`) : (Optional) The name of the collection in which to place the data item.

- itemProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs to use as the data item's properties.

- CreateDataNamespace()

  Purpose: Create a new data namespace. On successful creation, triggers the DataNamespaceCreation event.

  Command Data: dataName (Data Type) : Data description.

  - namespaceName (String) : The name of the data namespace.

- RemoveDataCollection()

  Purpose: Remove an existing data item collection from the given namespace. All data items in the collection will also be removed. On successful removal, triggers the DataCollectionRemoval event.

  Command Data: dataName (Data Type) : Data description.

  - collectionName (String) : The name of the data item collection.
  - collectionNamespace (String) : The name of the collection's namespace.

- RemoveDataItem()

  Purpose: Remove an existing data item from the given namespace and optional collection. Data item removal has no impact on any associated local file. On successful removal, triggers the DataItemRemoval event.

  Command Data: dataName (Data Type) : Data description.

  - itemName (String) : The name of the data item.
  - itemNamespace (String) : The name of the namespace from which to remove the data item.
  - itemCollection (String) : (Optional) The name of the collection from which to remove the data item.

- RemoveDataNamespace()

  Purpose: Remove an existing data namespace. All data items and collections in the namespace will also be removed. On successful removal, triggers the DataNamespaceRemoval event.

  Command Data: dataName (Data Type) : Data description.

  - namespaceName (String) : The name of the data namespace.

- UpdateDataItem()

  Purpose: Update the properties or details for an existing data item. For items created from byte buffers, only the properties can be updated. If the item is associated with a local file, the existing item's details are compared with the local file metadata and updated if the file contents have changed since the item was created (or last updated). On successful update, triggers the DataItemUpdate event.

Command Data: `dataName` (Data Type) : Data description.

- `itemName` (`String`) : The name of the data item.
- `itemNamespace` (`String`) : The name of the namespace in which the data item resides.
- `itemCollection` (`String`) : (Optional) The name of the collection in which the data item resides.
- `itemProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs to use to update the data item's properties.

Interactions: Request-Reply

- `GetDataCollectionDetails()`

  Purpose: Get the detailed information for the named data item collection.

  Request Data: `dataName` (Data Type) : Data description.

  - `collectionName` (`String`) : The name of the collection.
  - `collectionNamespace` (`String`) : The name of the namespace in which the collection resides.

  Reply Data: `dataName` (Data Type) : Data description.

  - `collectionDetails` (`DataCollectionDetails`) : The details for the collection.

- `GetDataItemAsBytes()`

  Purpose: Get the contents of the named data item as bytes. (NOTE: An implementation may limit the maximum size of a data item that can be fetched as bytes.)

  Request Data: `dataName` (Data Type) : Data description.

  - `itemName` (`String`) : The name of the data item.
  - `itemNamespace` (`String`) : The name of the namespace in which the data item resides.
  - `itemCollection` (`String`) : (Optional) The name of the collection in which the data item resides.

  Reply Data: `dataName` (Data Type) : Data description.

  - `contentBytes` (`Bytes`) : The data item contents.

- `GetDataItemAsLocalFile()`

  Purpose: Get the absolute path of a local file that can be used to retrieve the data item contents.

  Request Data: `dataName` (Data Type) : Data description.

  - `itemName` (`String`) : The name of the data item.

- `itemNamespace` (`String`) : The name of the namespace in which the data item resides.
- `itemCollection` (`String`) : (Optional) The name of the collection in which the data item resides.

Reply Data: `dataName` (Data Type) : Data description.

- `localFilePath` (`Filepath`) : The absolute path to a local file.
- `isTempFile` (`Boolean`) : Flag indicating whether the returned path is for a temporary file that should be deleted by the client when it is done using the file.

• `GetDataItemDetails()`

Purpose: Get the detailed information for the named data item.

Request Data: `dataName` (Data Type) : Data description.

- `itemName` (`String`) : The name of the data item.
- `itemNamespace` (`String`) : The name of the namespace in which the data item resides.
- `itemCollection` (`String`) : (Optional) The name of the collection in which the data item resides.

Reply Data: `dataName` (Data Type) : Data description.

- `itemDetails` (`DataItemDetails`) : The details for the data item.

• `GetDataNamespaceDetails()`

Purpose: Get the detailed information for the given namespace.

Request Data: `dataName` (Data Type) : Data description.

- `namespaceName` (`String`) : The name of the namespace.

Reply Data: `dataName` (Data Type) : Data description.

- `namespaceDetails` (`DataNamespaceDetails`) : The details for the namespace.

• `ListDataCollections()`

Purpose: Get a list of the data item collections that reside in the given namespace.

Request Data: `dataName` (Data Type) : Data description.

- `namespaceName` (`String`) : The name of the namespace to query for collections.

Reply Data: `dataName` (Data Type) : Data description.

- `collectionNames` (`List<String>`) : The list of collection names.

• `ListDataItems()`

Purpose: Get a list of the data items that reside in the given namespace and optional collection.

Request Data: `dataName` (Data Type) : Data description.

- `itemNamespace` (`String`) : The name of the namespace to query for data items.
- `itemCollection` (`String`) : (Optional) The name of the collection to query for data items.

Reply Data: `dataName` (Data Type) : Data description.

- `itemNames` (`List<String>`) : The list of data item names.

- `ListDataNamespaces()`

  Purpose: Get a list of the available namespaces.

  Reply Data: `dataName` (Data Type) : Data description.

  - `namespaceNames` (`List<String>`) : The list of namespaces.

Interactions: Asynchronous Status

- `DataCollectionCreation`

  Purpose: Notification of data item collection creations.

  Event Data: `dataName` (Data Type) : Data description.

  - `collectionDetails` (`DataCollectionDetails`) : The details for the collection.

- `DataCollectionRemoval`

  Purpose: Notification of data item collection removals.

  Event Data: `dataName` (Data Type) : Data description.

  - `collectionDetails` (`DataCollectionDetails`) : The details for the collection.

- `DataItemCreation`

  Purpose: Notification of data item creations.

  Event Data: `dataName` (Data Type) : Data description.

  - `itemDetails` (`DataItemDetails`) : The details for the data item.

- `DataItemRemoval`

  Purpose: Notification of data item removals.

  Event Data: `dataName` (Data Type) : Data description.

  - `itemDetails` (`DataItemDetails`) : The details for the data item.

- `DataItemUpdate`

  Purpose: Notification of data item updates.

  Event Data: `dataName` (Data Type) : Data description.

  - `itemDetails` (`DataItemDetails`) : The details for the data item.

- `DataNamespaceCreation`

  Purpose: Notification of data namespace creations.

  Event Data: `dataName` (Data Type) : Data description.

  - `namespaceDetails` (`DataNamespaceDetails`) : The details for the namespace.

- `DataNamespaceRemoval`

  Purpose: Notification of data item collection removals.

  Event Data: `dataName` (Data Type) : Data description.

  - `namespaceDetails` (`DataNamespaceDetails`) : The details for the namespace.

3.2.2.8 Capability: *Data Stream*

Description: Provides producer/consumer access to named streams of sequenced data items. A data stream can be considered a specialized form of data collection where each item in the collection is assigned a unique monotonically increasing index as its name when it is added to the collection. Stream data indices are numbered starting from 1. Consumers of the data stream may access its items in any order using the assigned indices, although the expectation is that sequential accesses will be most common.

Custom Data Type: `DataStreamDetails`

- `streamName` (`String`) : The name of the data item stream.
- `streamNamespace` (`String`) : The name of the namespace that contains the stream.
- `streamItemCount` (`Integer`) : The number of data items that have been added to the stream.
- `streamSize` (`Integer`) : The aggregate size in bytes of all the data items in the stream.
- `streamCreationTime` (`TimeStamp`) : Creation timestamp for the stream.
- `streamProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs for the stream's properties.

Related Capabilities: Extends

- *Data Storage*

Interactions: Command

- `CreateDataStream()`

  Purpose: Create a new data item stream in the given namespace. On successful creation, triggers the `DataStreamCreation` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `streamName` (`String`) : The name of the data item stream.
  - `streamNamespace` (`String`) : The name of the namespace in which to place the stream.
  - `streamProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs to use as the stream's properties.

- `AddDataStreamItem()`

  Purpose: Add an existing data item to the named stream. The item is a reference to the existing named item in the given namespace and optional collection, and thus no additional data is stored. On successful creation, triggers the `DataStreamItemAddition` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `itemName` (`String`) : The name of the existing data item.

- itemNamespace (String) : The name of the namespace in which the data item resides.
- itemCollection (String) : (Optional) The name of the collection in which the data item resides.
- streamName (String) : The name of the data item stream in which to put the new data item.
- streamNamespace (String) : The name of the stream's namespace.

- AddDataStreamCollection()

  Purpose: Adds all items from an existing data item collection to the named stream. The items from the existing named collection are added to the stream as references, and thus no additional data is stored. On successful creation, triggers the DataStreamItemAddition event.

  Command Data: dataName (Data Type) : Data description.

  - collectionName (String) : The name of the existing data item collection.
  - collectionNamespace (String) : The name of the namespace in which the collection resides.
  - streamName (String) : The name of the data item stream in which to put the collection's data items.
  - streamNamespace (String) : The name of the stream's namespace.

- AddDataStreamItemFromBytes()

  Purpose: Add a new data item with provided contents to the named stream. On successful creation, triggers the DataStreamItemAddition event. (NOTE: An implementation may limit the maximum size of a data item that can be created from bytes.)

  Command Data: dataName (Data Type) : Data description.

  - streamName (String) : The name of the data item stream in which to put the new data item.
  - streamNamespace (String) : The name of the stream's namespace.
  - contentBytes (Bytes) : The data item contents.
  - itemProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs to use as the data item's properties.

- AddDataStreamItemFromLocalFile()

  Purpose: Add a new data item associated with an existing local file to the named stream. On successful creation, triggers the DataStreamItemAddition event.

  Command Data: dataName (Data Type) : Data description.

  - `streamName` (`String`) : The name of the data item stream in which to put the new data item.
  - `streamNamespace` (`String`) : The name of the stream's namespace.
  - `localFilePath` (`Filepath`) : The absolute path of the local file from which to create the data item.
  - `itemProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs to use as the data item's properties.

- `RemoveDataStream()`

  Purpose: Remove an existing data item stream from the given namespace. All data items in the stream that are not references to existing data items will also be removed. On successful removal, triggers the `DataStreamRemoval` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `streamName` (`String`) : The name of the data item stream.
  - `streamNamespace` (`String`) : The name of the stream's namespace.

Interactions: Request-Reply

- `GetDataStreamDetails()`

  Purpose: Get the detailed information for the named data item collection.

  Request Data: `dataName` (Data Type) : Data description.

  - `streamName` (`String`) : The name of the data item stream.
  - `streamNamespace` (`String`) : The name of the namespace in which the stream resides.

  Reply Data: `dataName` (Data Type) : Data description.

  - `streamDetails` (`DataStreamDetails`) : The details for the data item stream.

- `GetDataStreamItemAsBytes()`

  Purpose: Get the contents of the named data item as bytes. (NOTE: An implementation may limit the maximum size of a data item that can be fetched as bytes.)

  Request Data: `dataName` (Data Type) : Data description.

  - `streamName` (`String`) : The name of the data item stream.
  - `streamNamespace` (`String`) : The name of the stream's namespace.
  - `itemIndex` (`Integer`) : The index of the data item within the named stream.

  Reply Data: `dataName` (Data Type) : Data description.

  - `contentBytes` (`Bytes`) : The data item contents.

- `GetDataStreamItemAsLocalFile()`

Purpose: Get the absolute path of a local file that can be used to retrieve the data item contents.

Request Data: `dataName` (Data Type) : Data description.

- `streamName` (`String`) : The name of the data item stream.
- `streamNamespace` (`String`) : The name of the stream's namespace.
- `itemIndex` (`Integer`) : The index of the data item within the named stream.

Reply Data: `dataName` (Data Type) : Data description.

- `localFilePath` (`Filepath`) : The absolute path to a local file.
- `isTempFile` (`Boolean`) : Flag indicating whether the returned path is for a temporary file that should be deleted by the client when it is done using the file.

• `GetDataStreamItemDetails()`

Purpose: Get the detailed information for the named data item.

Request Data: `dataName` (Data Type) : Data description.

- `streamName` (`String`) : The name of the data item stream.
- `streamNamespace` (`String`) : The name of the stream's namespace.
- `itemIndex` (`Integer`) : The index of the data item within the named stream.

Reply Data: `dataName` (Data Type) : Data description.

- `itemDetails` (`DataItemDetails`) : The details for the data item.

• `ListDataStreams()`

Purpose: Get a list of the data item streams that reside in the given namespace.

Request Data: `dataName` (Data Type) : Data description.

- `namespaceName` (`String`) : The name of the namespace to query for streams.

Reply Data: `dataName` (Data Type) : Data description.

- `streamNames` (`List<String>`) : The list of stream names.

Interactions: Asynchronous Status

• `DataStreamCreation`

Purpose: Notification of data item stream creations.

Event Data: `dataName` (Data Type) : Data description.

- `collectionDetails` (`DataCollectionDetails`) : The details for the data item stream.

• `DataStreamRemoval`

Purpose: Notification of data item stream removals.

Event Data: `dataName` (Data Type) : Data description.

- `streamDetails` (`DataStreamDetails`) : The details for the data item stream.

- `DataStreamItemAddition`

  Purpose: Notification of data item additions to a stream.

  Event Data: `dataName` (Data Type) : Data description.

  - `streamName` (`String`) : The name of the data item stream.
  - `streamNamespace` (`String`) : The name of the stream's namespace.
  - `itemIndex` (`Integer`) : The index of the most recently added data item.

3.2.2.9 Capability: *Data Transfer Orchestration*

Description: Orchestrates data product and stream transfers between *Data Storage* services using *Data Transfer Endpoint* agents.

Related Capabilities: Requires

- *Data Catalog*
- *Data Storage*
- *Data Stream*
- *Data Transfer Endpoint*

Interactions: Command

- `TransferDataProduct()`

  Purpose: Using the given transfer channel, transfer a data product (i.e., a data item or data item collection) from the source to destination according to the given transfer mode.

  Command Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `productId` (`UUID`) : The source catalog id of the data product to transfer.
  - `transferMode` (`String`) : The transfer mode (e.g., "COPY", "MOVE", or "MIRROR"). Copying duplicates the product at the destination, while moving entail removal of the source product. Mirroring is only supported for data item collections, and ensures that any data items added to the collection will be transferred.
  - `transferTimeout` (`Integer`) : (Optional) Timeout in seconds to complete the transfer (not valid for mirroring).

- `TransferDataStream()`

  Purpose: Using the given transfer channel, transfer a data item stream from the source to destination according to the given transfer mode.

  Command Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `streamId` (`UUID`) : The source catalog id of the data stream to transfer.
  - `transferMode` (`String`) : The transfer mode (e.g., "COPY" or "MIRROR"). Copying will transfer all items currently in the stream, while mirroring will only transfer new items added to the stream.

Interactions: Request-Reply

- `NegotiateDataTransfer()`

Purpose: Negotiate a data transfer channel between *Data Transfer Endpoint* agents to enable data product and data stream transfers between *Data Storage* services. On successful negotiation, triggers the `DataTransferChannelCreation` event.

Request Data: `dataName` (Data Type) : Data description.

- `srcCatalogService` (`UUID`) : The UUID of the source *Data Catalog* service.
- `srcStorageService` (`UUID`) : The UUID of the source *Data Storage* service.
- `dstStorageService` (`UUID`) : The UUID of the destination *Data Storage* service.
- `transferOptions` (`List<String>`) : A list of requested transfer options (e.g., "compression=on, transferBlock=4M"). There is no guarantee that any requested options will be used, as option availability dependencies on the negotiated transfer method and endpoint agent features.

Reply Data: `dataName` (Data Type) : Data description.

- `transferChannelId` (`Integer`) : A unique transfer channel identifier.
- `errorMsg` (`String`) : An error message describing why the transfer channel negotiation failed.

Interactions: Asynchronous Status

- `DataTransferChannelCreation`

Purpose: Notification of data transfer channel creations.

Event Data: `dataName` (Data Type) : Data description.

- `transferChannelId` (`Integer`) : A unique transfer channel identifier.
- `srcStorageService` (`UUID`) : The UUID of the source *Data Storage* service.
- `dstStorageService` (`UUID`) : The UUID of the destination *Data Storage* service.
- `srcTransferEndpoint` (`String`) : The channel endpoint information for the source *Data Transfer Endpoint* agent.
- `dstTransferEndpoint` (`String`) : The channel endpoint information for the destination *Data Transfer Endpoint* agent.
- `transferMethod` (`String`) : The transfer method used by the channel.

3.2.2.10 Capability: *Data Transfer Endpoint*

Description: Provides data transfer channels for data product and data stream transfers between *Data Storage* services.

Related Capabilities: Requires

- *Data Storage*
- *Data Stream*

Interactions: Command

- `SendDataItem()`

  Purpose: Send the named data item from the source storage service on the transfer channel. (NOTE: only valid for source endpoints)

  Command Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `itemName` (`String`) : The name of the data item to send on the transfer channel.
  - `itemNamespace` (`String`) : The name of the namespace in which the data item resides.
  - `itemCollection` (`String`) : [Optional] The name of the collection in which the data item resides.

- `ReceiveDataItem()`

  Purpose: Receive the named data item on the transfer channel and store it in the destination storage service. (NOTE: only valid for destination endpoints)

  Command Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `itemName` (`String`) : The name of the data item to send on the transfer channel.
  - `itemNamespace` (`String`) : The name of the namespace in which the data item resides.
  - `itemCollection` (`String`) : [Optional] The name of the collection in which the data item resides.

- `SendDataStreamItems()`

  Purpose: Send a sequence of data items denoted by first and last indices in the named stream from the source storage service on the transfer channel. (NOTE: only valid for source endpoints)

  Command Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.

- `streamName` (`String`) : The name of the data item stream.
- `streamNamespace` (`String`) : The name of the namespace in which the data item stream resides.
- `firstItemIndex` (`Integer`) : The stream index of the first data item in the sequence.
- `lastItemIndex` (`Integer`) : The stream index of the last data item in the sequence.

- `ReceiveDataStreamItems()`

  Purpose: Receive a sequence of data items denoted by first and last indices in the named stream on the transfer channel and add them to the corresponding stream in the destination storage service. (NOTE: only valid for destination endpoints)

  Command Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `streamName` (`String`) : The name of the data item stream.
  - `streamNamespace` (`String`) : The name of the namespace in which the data item stream resides.
  - `firstItemIndex` (`Integer`) : The stream index of the first data item in the sequence.
  - `lastItemIndex` (`Integer`) : The stream index of the last data item in the sequence.

Interactions: Request-Reply

- `GetSupportedTransferMethods()`

  Purpose: Get a list of the supported data transfer methods.

  Reply Data: `dataName` (Data Type) : Data description.

  - `transferMethods` (`List< String >`) : A list of supported data transfer methods.

- `CreateDataTransferSource()`

  Purpose: Using the given transfer channel id, transfer method, and transfer channel options, establish a data transfer channel and return its source endpoint information. On successful creation, triggers the `DataTransferSourceEndpointCreation` event.

  Request Data: `dataName` (Data Type) : Data description.

  - `srcStorageService` (`UUID`) : The UUID of the source *Data Storage* service.
  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `transferMethod` (`String`) : The transfer method to use for the channel endpoint.
  - `transferOptions` (`List<String>`) : The transfer options requested for the channel.

  Reply Data: `dataName` (Data Type) : Data description.

  - `endpointInfo` (`String`) : The information needed by the destination endpoint to establish a channel connection to the source endpoint.

- `transferOptions` (`List<String>`) : The transfer options used for the channel.
- `errorMsg` (`String`) : An error message describing why the transfer channel endpoint could not be established.

- `CreateDataTransferDestination()`

  Purpose: Using the given transfer channel id, transfer method, and transfer channel options, establish a data transfer channel and return its destination endpoint information. On successful creation, triggers the `DataTransferDestinationEndpointCreation` event.

  Request Data: `dataName` (Data Type) : Data description.

  - `dstStorageService` (`UUID`) : The UUID of the destination *Data Storage* service.
  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `transferMethod` (`String`) : The transfer method to use for the channel endpoint.
  - `transferOptions` (`List<String>`) : The transfer options requested for the channel.

  Reply Data: `dataName` (Data Type) : Data description.

  - `endpointInfo` (`String`) : The information needed by the source endpoint to establish a channel connection to the destination endpoint.
  - `transferOptions` (`List<String>`) : The transfer options used for the channel.
  - `errorMsg` (`String`) : An error message describing why the transfer channel endpoint could not be established.

Interactions: Asynchronous Status

- `DataTransferSourceEndpointCreation`

  Purpose: Notification of data transfer channel source endpoint creations.

  Event Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.
  - `srcStorageService` (`UUID`) : The UUID of the source *Data Storage* service.
  - `srcTransferEndpoint` (`String`) : The channel endpoint information for the source *Data Transfer Endpoint* agent.
  - `transferMethod` (`String`) : The transfer method used by the channel.
  - `transferOptions` (`List<String>`) : The transfer options used by the channel.

- `DataTransferDestinationEndpointCreation`

  Purpose: Notification of data transfer channel destination endpoint creations.

  Event Data: `dataName` (Data Type) : Data description.

  - `transferChannelId` (`Integer`) : A unique transfer channel identifier.

– `dstStorageService` (`UUID`) : The UUID of the destination *Data Storage* service.

– `dstTransferEndpoint` (`String`) : The channel endpoint information for the destination *Data Transfer Endpoint* agent.

– `transferMethod` (`String`) : The transfer method used by the channel.

– `transferOptions` (`List<String>`) : The transfer options used by the channel.

3.2.3 INTERSECT System Management

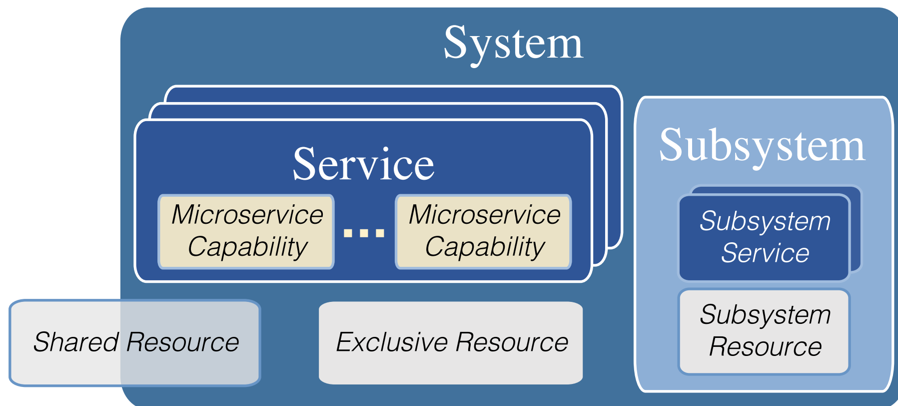

Figure 3-3. INTERSECT System, Subsystem, Service, and Resource Hierarchy

As shown in Figure 3-3, a *System* within INTERSECT consists of one or more *Services* and a collection of associated *Resources*. Each service provides utility in the form of a set of microservice capabilities. All INTERSECT activities involving system resources are facilitated through service interactions. A given resource may be exclusive to a system or shared amongst systems. An *Exclusive Resource* is one that is only accessed by the parent system's services. A *Shared Resource* may be accessed by services from other systems.

A system may also include *Subsystems*, which are self-contained systems that are used by the parent system. Subsystems typically exist to maintain operational independence over a group of services that provide access and control of one or more system resources.

System management services broadly represent the ability to control and inspect systems, subsystems, and resources. Examples of systems include, but are not limited to:

- computing systems (e.g., cloud, edge, or high-performance computing (HPC) systems)
- data and information systems (e.g., file systems, databases, and data catalogs)
- industrial control systems (e.g., programmable logic controllers and robotics)
- scientific instruments (e.g., electron microscopes and neutron detectors)

To enable dynamic interconnected science ecosystems, each INTERSECT system is expected manage its associated services, resources, and subsystems for its entire useful lifetime. It is therefore assumed that every INTERSECT system will have an associated *System Management* service that coordinates all aspects related to system information management, control of services and subsystems, and status monitoring of associated resources, services, and subsystems. When a system is newly introduced to an ecosystem, it must register itself with the ecosystem and provide contact details for its *System Management* service so that other members of the ecosystem may obtain information on the system's underlying services, resources, and subsystems. Additional details on the expected microservice interactions related to system, service, and resource registration can be found in Appendix A-1.

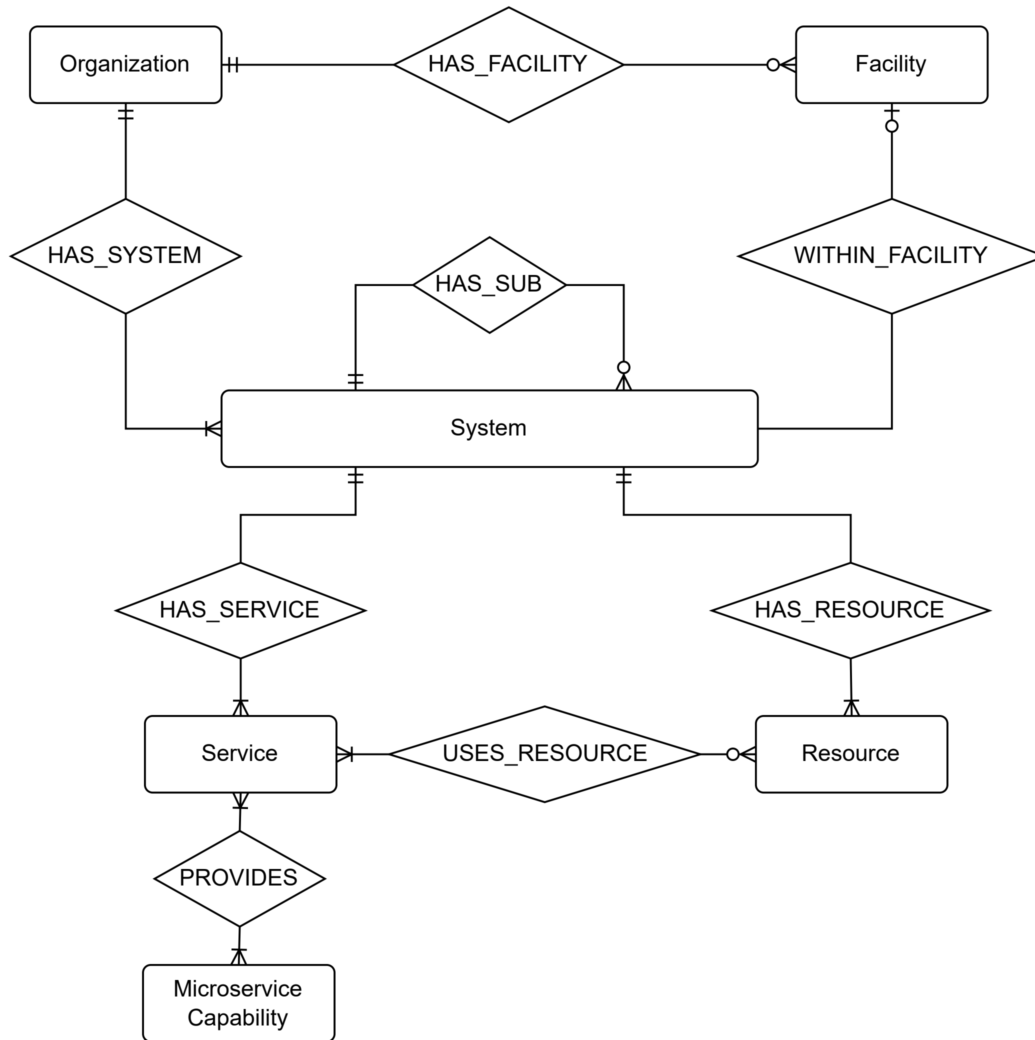

Figure 3-4. INTERSECT System Entity-Relation Model

3.2.3.1 Capability: *System Information Catalog*

Description: Provides subsystem, service, and resource information for a single parent system. Maintains system relationships as shown in Figure 3-4.

Related Capabilities: Requires

- *Entity-Relationship Catalog*

Interactions: Command

- `CreateSystemResource()`

  Purpose: Creates a new resource of the parent system. Successful creation will result in the creation of a `SystemResource` entity with the given information and a relationship to the new entity with the parent system. Figure A-3 provides an example orchestration sequence demonstrating the use of this method.

  Command Data: `dataName` (Data Type) : Data description.

  – `resourceDescription` (`String`) : A user-friendly description of the resource.

  – `resourceId` (`UUID`) : The UUID of the resource.

  – `resourceName` (`String`) : A user-friendly name for the resource.

  – `resourceLabels` (`List<String>`) : (Optional) A list of `String` labels for the resource.

  – `resourceProperties` (`List< KeyVal<String> >`) : (Optional) A list of key-value `String` pairs for the resource's properties.

- `CreateSystemService()`

  Purpose: Creates a new service of the parent system or an existing subsystem. Successful creation will result in the creation of a `SystemService` entity with the given information and relationships to the new entity with the parent system or subsystem, provided capabilities, and any associated resources. Figure A-4 provides an example orchestration sequence demonstrating the use of this method.

  Command Data: `dataName` (Data Type) : Data description.

  – `serviceDescription` (`String`) : A user-friendly description of the service.

  – `serviceId` (`UUID`) : The UUID of the service.

  – `serviceName` (`String`) : A user-friendly name for the service.

  – `serviceCapabilities` (`List<String>`) : A list of names for the microservice capabilities provided by the service.

  – `subsystemId` (`UUID`) : (Optional) The UUID of the subsystem in which this service exists.

  – `serviceLabels` (`List<String>`) : (Optional) A list of `String` labels for the service.

- servicePropertiesParams

– serviceProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs for the service's properties.

– serviceResources (List<UUID>) : (Optional) A list of system resources used by the service.

- CreateSubsystem()

Purpose: Creates a new subsystem of the parent system. Successful creation will result in the creation of a System entity with the given information and relationships to the new entity with the parent system and any associated resources.

Command Data: dataName (Data Type) : Data description.

– subsystemDescription (String) : A user-friendly description of the subsystem.

– subsystemId (UUID) : The UUID of the subsystem.

– subsystemName (String) : The name of the subsystem.

– subsystemLabels (List<String>) : (Optional) A list of String labels for the entity.

– subsystemProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs for the entity's properties.

– subsystemResources (List<UUID>) : (Optional) A list of system resources associated with the subsystem.

Interactions: Request-Reply

- GetSubsystemInformation()

Purpose: Get the full set of system catalog information for the given subsystemId or subsystemName.

Request Data: dataName (Data Type) : Data description.

– subsystemId (UUID) : The UUID of the subsystem.

– subsystemName (String) : The name of the subsystem.

Reply Data: dataName (Data Type) : Data description.

– subsystemInfo (CatalogEntityInformation) : The catalog information for the subsystem.

– errorMsg (String) : (Optional) An error message describing why the lookup failed.

- GetSubsystemIds()

Purpose: Get a list of subsystem UUIDs for the parent system.

Reply Data: dataName (Data Type) : Data description.

– subsystemIds (List<UUID>) : The list of subsystem UUIDs.

– errorMsg (String) : (Optional) An error message describing why the lookup failed.

- GetSubsystemNames()

  Purpose: Get a list of subsystem names for the parent system.

  Reply Data: dataName (Data Type) : Data description.

  – subsystemNames (List<String>) : The list of subsystem names.

  – errorMsg (String) : (Optional) An error message describing why the lookup failed.

- GetSystemResourceInformation()

  Purpose: Get the full set of system catalog information for the given resourceId or resourceName.

  Request Data: dataName (Data Type) : Data description.

  – resourceId (UUID) : The UUID of the resource.

  – resourceName (String) : The name of the resource.

  Reply Data: dataName (Data Type) : Data description.

  – resourceInfo (CatalogEntityInformation) : The catalog information for the resource.

  – errorMsg (String) : (Optional) An error message describing why the lookup failed.

- GetSystemResourceIds()

  Purpose: Get a list of resource UUIDs for the parent system or given subsystemName.

  Request Data: dataName (Data Type) : Data description.

  – subsystemName (String) : (Optional) The name of the subsystem.

  Reply Data: dataName (Data Type) : Data description.

  – resourceIds (List<UUID>) : The list of system resource UUIDs.

  – errorMsg (String) : (Optional) An error message describing why the lookup failed.

- GetSystemResourceNames()

  Purpose: Get a list of resource names for the parent system or given subsystemName.

  Request Data: dataName (Data Type) : Data description.

  – subsystemName (String) : (Optional) The name of the subsystem.

  Reply Data: dataName (Data Type) : Data description.

  – resourceNames (List<String>) : The list of system resource names.

  – errorMsg (String) : (Optional) An error message describing why the lookup failed.

- GetSystemServiceInformation()

Purpose: Get the full set of system catalog information for the given `serviceId` or `serviceName`.

Request Data: `dataName` (Data Type) : Data description.

- `serviceId` (`UUID`) : The UUID of the service.
- `serviceName` (`String`) : The name of the service.

Reply Data: `dataName` (Data Type) : Data description.

- `serviceInfo` (`CatalogEntityInformation`) : The catalog information for the service.
- `errorMsg` (`String`) : (Optional) An error message describing why the lookup failed.

• `GetSystemServiceIds()`

Purpose: Get a list of service UUIDs for the parent system or the given `subsystemName`.

Request Data: `dataName` (Data Type) : Data description.

- `subsystemName` (`String`) : (Optional) The name of the subsystem.

Reply Data: `dataName` (Data Type) : Data description.

- `serviceIds` (`List<UUID>`) : The list of system service UUIDs.
- `errorMsg` (`String`) : (Optional) An error message describing why the lookup failed.

• `GetSystemServiceNames()`

Purpose: Get a list of service names for the parent system or the given `subsystemName`.

Request Data: `dataName` (Data Type) : Data description.

- `subsystemName` (`String`) : (Optional) The name of the subsystem.

Reply Data: `dataName` (Data Type) : Data description.

- `serviceNames` (`List<String>`) : The list of system service names.
- `errorMsg` (`String`) : (Optional) An error message describing why the lookup failed.

• `GetSystemServicesByCapability()`

Purpose: Get a list of UUIDs for services in the parent system or given `subsystemName` that provide the microservice capability with given `capabilityName`.

Request Data: `dataName` (Data Type) : Data description.

- `capabilityName` (`String`) : The name of the desired capability.
- `subsystemName` (`String`) : (Optional) The name of the subsystem.

Reply Data: `dataName` (Data Type) : Data description.

- `serviceIds` (`List<UUID>`) : The list of matching system service UUIDs.

- errorMsg (String) : (Optional) An error message describing why the lookup failed.

- GetSystemServicesByResource()

  Purpose: Get a list of UUIDs for services in the parent system or given subsystemName that use the resource with given resourceName.

  Request Data: dataName (Data Type) : Data description.

  - resourceName (String) : The name of the resource.
  - subsystemName (String) : (Optional) The name of the subsystem.

  Reply Data: dataName (Data Type) : Data description.

  - serviceIds (List<UUID>) : The list of matching system service UUIDs.
  - errorMsg (String) : (Optional) An error message describing why the lookup failed.

3.2.3.2 Capability: *System Manager*

Description: Provides interfaces for aggregate control and status of all subsystems, resources, and services for a single parent system. Uses the *Availability Status* capability of each system service.

Related Capabilities: Requires

- *System Information Catalog*
- *Availability Status*

Interactions: Command

- `EnableResource()`

  Purpose: Enable the target resource identified by `resourceId` or `resourceName` in the parent system. Any services or subsystems associated with the resource will have their status updated accordingly.

  Command Data: `dataName` (Data Type) : Data description.

  - `resourceId` (`UUID`) : (Optional) The UUID of the resource.
  - `resourceName` (`String`) : (Optional) The name of the resource.
  - `statusNote` (`String`) : (Optional) An optional short description to use when updating the resource status.

- `DisableResource()`

  Purpose: Disable the target resource identified by `resourceId` or `resourceName` in the parent system. Any services or subsystems associated with the resource will have their status updated accordingly.

  Command Data: `dataName` (Data Type) : Data description.

  - `resourceId` (`UUID`) : (Optional) The UUID of the resource.
  - `resourceName` (`String`) : (Optional) The name of the resource.
  - `statusNote` (`String`) : (Optional) An optional short description to use when updating the resource status.

- `EnableService()`

  Purpose: Enable the target service identified by `serviceId` or `serviceName` in the parent system. If the service is associated with a subsystem, its status will be updated accordingly.

  Command Data: `dataName` (Data Type) : Data description.

  - `serviceId` (`UUID`) : (Optional) The UUID of the service.
  - `serviceName` (`String`) : (Optional) The name of the service.
  - `statusNote` (`String`) : (Optional) An optional short description to use when updating the service status.

- `DisableService()`

  Purpose: Disable the target service identified by `serviceId` or `serviceName` in the parent system. If the service is associated with a subsystem, its status will be updated accordingly.

  Command Data: `dataName` (Data Type) : Data description.

  - `serviceId` (`UUID`) : (Optional) The UUID of the service.
  - `serviceName` (`String`) : (Optional) The name of the service.
  - `statusNote` (`String`) : (Optional) An optional short description to use when updating the service status.

- `EnableSubsystem()`

  Purpose: Enable all services in the target subsystem identified by `subsystemId` or `subsystemName`.

  Command Data: `dataName` (Data Type) : Data description.

  - `subsystemId` (`UUID`) : (Optional) The UUID of the subsystem.
  - `subsystemName` (`String`) : (Optional) The name of the subsystem.
  - `statusNote` (`String`) : (Optional) An optional short description to use when updating the status of the subsystem services.

- `DisableSubsystem()`

  Purpose: Disable all services in the target subsystem identified by `subsystemId` or `subsystemName`.

  Command Data: `dataName` (Data Type) : Data description.

  - `subsystemId` (`UUID`) : (Optional) The UUID of the subsystem.
  - `subsystemName` (`String`) : (Optional) The name of the subsystem.
  - `statusNote` (`String`) : (Optional) An optional short description to use when updating the status of the subsystem services.

- `EnableSystem()`

  Purpose: Enable all subsystems, services, and resources in the parent system.

  Command Data: `dataName` (Data Type) : Data description.

  - `statusNote` (`String`) : (Optional) An optional short description to use when updating the system status.

- `DisableSystem()`

  Purpose: Disable all subsystems, services, and resources in the parent system.

  Command Data: `dataName` (Data Type) : Data description.

- statusNote (String) : (Optional) An optional short description to use when updating the system status.

Interactions: Request-Reply

- RegisterService()

  Purpose: Registers a new service with the target system and an optional subsystem. Figure A-4 provides an example orchestration sequence demonstrating the use of this method.

  Request Data: dataName (Data Type) : Data description.

  - serviceName (String) : A user-friendly name for the service.
  - serviceDescription (String) : A user-friendly description of the service.
  - serviceCapabilities (List<String>) : A list of names for the microservice capabilities provided by the service.
  - subsystemId (UUID) : (Optional) The UUID of the subsystem in which this service exists.
  - serviceResources (List<UUID>) : (Optional) A list of system resources used by the service.
  - serviceLabels (List<String>) : (Optional) A list of String labels for the service.
  - serviceProperties (List< KeyVal<String> >) : (Optional) A list of key-value String pairs for the service's properties.

  Reply Data: dataName (Data Type) : Data description.

  - serviceUUID (String) : The UUID assigned to the service by the system.
  - errorMsg (String) : (Optional) An error message describing why the service could not be registered.

- GetResourceStatus()

  Purpose: Get the current status of the target resource identified by resourceId or resourceLabelsceName in the parent system.

  Request Data: dataName (Data Type) : Data description.

  - resourceId (UUID) : (Optional) The UUID of the resource.
  - resourceName (String) : (Optional) The name of the resource.

  Reply Data: dataName (Data Type) : Data description.

  - resourceStatus (String) : The current status of the resource.
  - errorMsg (String) : (Optional) An error message describing why the resource status could not be determined.

- GetServiceStatus()

Purpose: Get the current status of the target service identified by `serviceId` or `serviceName` in the parent system.

Request Data: `dataName` (Data Type) : Data description.

- `serviceId` (`UUID`) : (Optional) The UUID of the service.
- `serviceName` (`String`) : (Optional) The name of the service.

Reply Data: `dataName` (Data Type) : Data description.

- `serviceStatus` (`String`) : The current status of the service.
- `errorMsg` (`String`) : (Optional) An error message describing why the service status could not be determined.

• `GetSubsystemStatus()`

Purpose: Gather and aggregate the status of all services in the subsystem identified by `subsystemId` or `subsystemName`.

Request Data: `dataName` (Data Type) : Data description.

- `subsystemId` (`UUID`) : (Optional) The UUID of the subsystem.
- `subsystemName` (`String`) : (Optional) The name of the subsystem.

Reply Data: `dataName` (Data Type) : Data description.

- `subsystemStatus` (`String`) : The aggregate status of the subsystem.
- `errorMsg` (`String`) : (Optional) An error message describing why the subsystem status could not be determined.

• `GetSystemStatus()`

Purpose: Gather and aggregate the status of all services and subsystems in the parent system.

Reply Data: `dataName` (Data Type) : Data description.

- `systemStatus` (`String`) : The aggregate status of the subsystem.
- `errorMsg` (`String`) : (Optional) An error message describing why the aggregate status could not be determined.

Interactions: Asynchronous Status

• `SystemStatusChange`

Purpose: Notification of changes to system status.

Event Data: `dataName` (Data Type) : Data description.

- `systemId` (`UUID`) : The UUID of the system.
- `currentStatus` (`String`) : The current system status.
- `changeDescription` (`String`) : A description of the last status change.

- `changeTime` (`Timestamp`) : A timestamp for the last status change.
- `priorStatus` (`String`) : (Optional) The prior system status.
- `subsystemId` (`UUID`) : (Optional) The UUID of the subsystem, if applicable.
- `resourceId` (`UUID`) : (Optional) The UUID of the resource, if applicable.
- `serviceId` (`UUID`) : (Optional) The UUID of the service, if applicable.

3.2.3.3 Capability: *Systems Registrar*

Description: Provides system, subsystem, service, and resource registration and name-based lookup of assigned UUIDs.

Related Capabilities: Requires

- *UUID Generation*

Interactions: Request-Reply

- `RegisterSystem()`

  Purpose: Register a new system with given `systemName`, `organizationName`, and optional `facilityName`. Successful registration will trigger generation of the `SystemRegistration` event. Figure A-2 provides an example orchestration sequence demonstrating the use of this method.

  Request Data: `dataName` (Data Type) : Data description.

  - `systemName` (`String`) : The name of the system.
  - `organizationName` (`String`) : The name of the organization that owns the system.
  - `facilityName` (`String`) : (Optional) The name of the user facility or sub-organization that manages operation of the system.
  - `systemSecret` (`Bytes`) : (Optional) A system secret for use in verifying registrations of system components.
  - `requestedId` (`UUID`) : (Optional) The requested UUID for the system.

  Reply Data: `dataName` (Data Type) : Data description.

  - `systemId` (`UUID`) : The assigned UUID for the system, or an `INVALID_UUID` value (e.g., all zeroes) if the registration failed.
  - `errorMsg` (`String`) : (Optional) An error message describing why the system could not be successfully registered.

- `GetSystemUUID()`

  Purpose: Get the assigned UUID for the given `systemName`, `organizationName`, and optional `facilityName`.

  Request Data: `dataName` (Data Type) : Data description.

  - `systemName` (`String`) : The name of the system.
  - `organizationName` (`String`) : The name of the organization that owns the system.
  - `facilityName` (`String`) : (Optional) The name of the user facility or sub-organization that manages operation of the system.

  Reply Data: `dataName` (Data Type) : Data description.

- `systemId` (`UUID`) : The assigned UUID for the system, or an `INVALID_UUID` value (e.g., all zeroes) if the lookup failed.
- `errorMsg` (`String`) : (Optional) An error message describing why the system lookup failed.

- `RegisterSubsystem()`

  Purpose: Register a new subsystem of the given parent system. Successful registration will trigger generation of the `SubsystemRegistration` event.

  Request Data: `dataName` (Data Type) : Data description.

  - `subsystemName` (`String`) : The name of the subsystem.
  - `systemId` (`UUID`) : The UUID of the parent system.
  - `systemSecret` (`Bytes`) : (Optional) Secret key for the system for use in verifying registrations of system components.
  - `requestedId` (`UUID`) : (Optional) The requested UUID for the subsystem.

  Reply Data: `dataName` (Data Type) : Data description.

  - `subsystemId` (`UUID`) : The assigned UUID for the subsystem, or an `INVALID_UUID` value (e.g., all zeroes) if the registration failed.
  - `errorMsg` (`String`) : (Optional) An error message describing why the subsystem could not be successfully registered.

- `GetSubsystemUUID()`

  Purpose: Get the assigned UUID for the given `subsystemName`.

  Request Data: `dataName` (Data Type) : Data description.

  - `systemId` (`UUID`) : The UUID of the parent system.
  - `subsystemName` (`String`) : The name of the subsystem.

  Reply Data: `dataName` (Data Type) : Data description.

  - `subsystemId` (`UUID`) : The assigned UUID for the subsystem, or an `INVALID_UUID` value (e.g., all zeroes) if the lookup failed.
  - `errorMsg` (`String`) : (Optional) An error message describing why the subsystem lookup failed.

- `RegisterSystemResource()`

  Purpose: Register a new resource within the given parent system. Successful registration will trigger generation of the `SystemResourceRegistration` event. Figure A-3 provides an example orchestration sequence demonstrating the use of this method.

  Request Data: `dataName` (Data Type) : Data description.

  - `resourceName` (`String`) : The name of the system resource.

- `systemId` (`UUID`) : The UUID of the parent system.
- `systemSecret` (`Bytes`) : (Optional) Secret key for the system for use in verifying registrations of system components.
- `requestedId` (`UUID`) : (Optional) The requested UUID for the resource.

Reply Data: `dataName` (Data Type) : Data description.

- `resourceId` (`UUID`) : The assigned UUID for the system resource, or an `INVALID_UUID` value (e.g., all zeroes) if the registration failed.
- `errorMsg` (`String`) : (Optional) An error message describing why the system resource could not be successfully registered.

• `GetSystemResourceUUID()`

Purpose: Get the assigned UUID for the given `resourceName`,.

Request Data: `dataName` (Data Type) : Data description.

- `systemId` (`UUID`) : The UUID of the parent system.
- `resourceName` (`String`) : The name of the system resource.

Reply Data: `dataName` (Data Type) : Data description.

- `resourceId` (`UUID`) : The assigned UUID for the system resource, or an `INVALID_UUID` value (e.g., all zeroes) if the lookup failed.
- `errorMsg` (`String`) : (Optional) An error message describing why the system resource lookup failed.

• `RegisterSystemService()`

Purpose: Register a new service within the given parent system, with optional association to a given subsystem. Successful registration will trigger generation of the `SystemServiceRegistration` event. Figure A-4 provides an example orchestration sequence demonstrating the use of this method.

Request Data: `dataName` (Data Type) : Data description.

- `serviceName` (`String`) : The name of the system service (e.g., `<hostname>.<pid>`).
- `systemId` (`UUID`) : The UUID of the parent system.
- `subsystemId` (`UUID`) : (Optional) The UUID of the associated subsystem.
- `systemSecret` (`Bytes`) : (Optional) Secret key for the system for use in verifying registrations of system components.
- `requestedId` (`UUID`) : (Optional) The requested UUID for the service.

Reply Data: `dataName` (Data Type) : Data description.

- `serviceId` (`UUID`) : The assigned UUID for the system service, or an `INVALID_UUID` value (e.g., all zeroes) if the registration failed.

- `errorMsg` (`String`) : (Optional) An error message describing why the system service could not be successfully registered.

- `GetSystemServiceUUID()`

  Purpose: Get the assigned UUID for the given `serviceName`,.

  Request Data: `dataName` (Data Type) : Data description.

  - `systemId` (`UUID`) : The UUID of the parent system.
  - `subsystemId` (`UUID`) : (Optional) The UUID of the associated subsystem.
  - `serviceName` (`String`) : The name of the system service.

  Reply Data: `dataName` (Data Type) : Data description.

  - `serviceId` (`UUID`) : The assigned UUID for the system service, or an `INVALID_UUID` value (e.g., all zeroes) if the lookup failed.
  - `errorMsg` (`String`) :(Optional) An error message describing why the system service lookup failed.

Interactions: Asynchronous Status

- `SystemRegistration`

  Purpose: Notification of new system registrations.

  Event Data: `dataName` (Data Type) : Data description.

  - `systemId` (`UUID`) : The assigned UUID for the system.
  - `systemName` (`String`) : The name of the system.
  - `organizationName` (`String`) : The name of the organization that owns the system.
  - `facilityName` (`String`) : (Optional) The name of the user facility or sub-organization that manages operation of the system.

- `SubsystemRegistration`

  Purpose: Notification of new subsystem registrations.

  Event Data: `dataName` (Data Type) : Data description.

  - `systemId` (`UUID`) : The parent system UUID.
  - `subsystemId` (`String`) : The assigned UUID of the subsystem.
  - `subsystemName` (`String`) : The name of the subsystem.

- `SystemResourceRegistration`

  Purpose: Notification of new system resource registrations.

  Event Data: `dataName` (Data Type) : Data description.

  - `systemId` (`UUID`) : The parent system UUID.

- `resourceId` (`UUID`) : The assigned system resource UUID.
- `resourceName` (`String`) : The name of the system resource.

- `SystemServiceRegistration`

  Purpose: Notification of new system service registrations.

  Event Data: `dataName` (Data Type) : Data description.

  - `systemId` (`UUID`) : The parent system UUID.
  - `subsystemId` (`UUID`) : The subsystem UUID, or an `INVALID_UUID` value (e.g., all zeroes) if the service is not associated to a subsystem.
  - `serviceId` (`UUID`) : The assigned system service UUID.
  - `serviceName` (`String`) : The name of the system service.

3.2.4 INTERSECT Resource Adapters

3.2.4.1 Compute Adapters

Compute adapters broadly represent the ability to leverage local or remote computational resources to process data or run applications. For the INTERSECT Open Architecture, four types of computational resources are expected to be employed: (1) HPC systems, (2) cloud computing systems, (3) edge computing systems, and (4) individual host computers. HPC systems provide large-scale computational support for scientific modeling and simulation, high-throughput processing, and model training for AI using high-performance compute, storage, and networking hardware. Cloud computing systems provide general-purpose computational support using commodity server-based compute, storage, and networking hardware. A given cloud computing system may exist within an organization (i.e., a private cloud) or be publicly available on the Internet (i.e., a public cloud). Edge computing systems provide computational support for low latency processsing of data produced by nearby sources (e.g., sensors or scientific instruments) using a variety of hardware (e.g., high-performance, commodity, or embedded). For host computing, the designated host may exist within an HPC, cloud, or edge computing system.

There are two computing abstractions commonly provided by compute resources: batch computing or on-demand computing. Batch computing provides adapters to batch job systems (e.g., IBM Spectrum Load Sharing Facility (LSF), Portable Batch System (PBS), and Simple Linux Utility for Resource Management (SLURM)) to schedule, run, and monitor jobs across one or more job queues. Because batch computing systems are shared amongst many concurrent users with varied resource requirements and scheduling priorities, batch jobs have unpredictable completion latencies. Jobs may be queued for several hours or even days before being allocated resources to run. In contrast, on-demand computing provides instantaneous allocation when the requested resources are available, or the request will be immediately rejected. On-demand computing is currently not widely supported by HPC systems, but is readily available in cloud or edge environments.

3.2.4.2 Capability: *Compute Allocation*

Description: Interact with computing systems to allocate immediate access to a set of compute hosts for a specified period of time. Figure A-6 shows an example orchestration seqence for on-demand allocation of computing resources. Figure A-7 shows an example orchestration seqence for running an application program within an allocation of computing resources.

Interactions: Command

- ReleaseAllocation()

  Purpose: Release an existing compute system allocation.

  Command Data: dataName (Data Type) : Data description.

  - allocationId (String) : The compute system allocation identifier.

- RunInAllocation()

  Purpose: Run a program using the given run options and program arguments within an existing compute system allocation.

  Command Data: dataName (Data Type) : Data description.

  - runName (String) : The name for the run.
  - allocationId (String) : The compute system allocation identifier.
  - runOptions (List<String>) : The run options for the compute resource manager. The format of the provided options is unspecified and may vary across compute resource managers.
  - programArgs (List<String>) : The full program arguments. The first argument in the list should be the local file path to the program executable.

Interactions: Request-Reply

- CreateAllocation()

  Purpose: Create a compute system allocation.

  Request Data: dataName (Data Type) : Data description.

  - hostCount (Integer) : The number of compute hosts to allocate.
  - wallTime (TimeDuration) : The requested walltime for the allocation.
  - featureMatchCriteria (List< KeyVal<String> >) : An optional list of key-value parameters to use in selection of compute system hosts with desired features.

  Reply Data: dataName (Data Type) : Data description.

  - allocationId (String) : The compute system allocation identifier.

- errorMsg (String) : (Optional) An error message describing why the requested allocation could not be granted.

- GetAllocationDetails()

  Purpose: Request detailed allocation information as a list of key-value pairs for the given allocation, such as the allocation start and end times and resources included within the allocation. The format of the keys and values in the returned information is unspecified and may vary across computing systems.

  Request Data: dataName (Data Type) : Data description.

  - allocationId (String) : The compute system allocation identifier.

  Reply Data: dataName (Data Type) : Data description.

  - allocationDetails (List< KeyVal<String> >) : The detailed allocation information for the given allocationId.
  - errorMsg (String) : (Optional) An error message describing why the requested compute system allocation information could not be returned for the given allocationId.

- GetAllocationInfo()

  Purpose: Request summary information for the given allocation, such as the set of allocated compute hosts and their features. The format of the returned information is unspecified and may vary across computing systems.

  Request Data: dataName (Data Type) : Data description.

  - allocationId (String) : The compute system allocation identifier.

  Reply Data: dataName (Data Type) : Data description.

  - allocationInfo (String) : The summary allocation information for the given allocationId.
  - errorMsg (String) : (Optional) An error message describing why the requested compute system allocation information could not be returned for the given allocationId.

- GetAllocationStatus()

  Purpose: Request current status for the given allocation.

  Request Data: dataName (Data Type) : Data description.

  - allocationId (String) : The compute system allocation identifier.

  Reply Data: dataName (Data Type) : Data description.

  - allocationStatus (String) : The compute system allocation status information for the given allocationId (e.g., "ALLOCATION_ACTIVE", or "ALLOCATION_ENDED").

- errorMsg (String) : (Optional) An error message describing why the requested compute system allocation status could not be returned for the given allocationId.

- GetRunDetails()

  Purpose: Request detailed information as a list of key-value pairs for the run associated with the given name. The format of the keys and values in the returned information is unspecified and may vary across computing systems.

  Request Data: dataName (Data Type) : Data description.

  - runName (String) : The name of the run.

  Reply Data: dataName (Data Type) : Data description.

  - runDetails (List< KeyVal<String> >) : The detailed run information for the given runName.
  - errorMsg (String) : (Optional) An error message describing why the requested compute system run information could not be returned for the given runName.

- GetRunStatus()

  Purpose: Request current status for the run associated with the given name.

  Request Data: dataName (Data Type) : Data description.

  - runName (String) : The name of the run.

  Reply Data: dataName (Data Type) : Data description.

  - runStatus (String) : The compute system alloruncation status information for the given runName (e.g., "RUN_IN_PROGRESS", "RUN_FAILED", or "RUN_COMPLETED").
  - errorMsg (String) : (Optional) An error message describing why the requested compute system run status could not be returned for the given runName.

Interactions: Asynchronous Status

- ComputeAllocationStatusChange

  Purpose: Notification of compute system allocation status changes.

  Event Data: dataName (Data Type) : Data description.

  - allocationId (String) : The compute system allocation identifier.
  - allocationStatus (String) : The compute system allocation status information for the given allocationId (e.g., "ALLOCATION_ACTIVE", "ALLOCATION_ENDED", or "ALLOCATION_EXPIRED").
  - statusDetails (String) : (Optional) Additional information further detailing the current status.

- ComputeRunStatusChange

Purpose: Notification of compute system allocation status changes.

Event Data: `dataName` (Data Type) : Data description.

- `allocationId` (`String`) : The identifier of the compute system allocation containing the run.
- `runName` (`String`) : The name of the run.
- `runStatus` (`String`) : The compute system run status information for the given `runName` (e.g., "RUN_IN_PROGRESS", "RUN_FAILED", or "RUN_COMPLETED").
- `statusDetails` (`String`) : (Optional) Additional information further detailing the current status.

3.2.4.3 Capability: *Compute Queue*

Description: Interact with batch job queueing systems to run compute jobs and monitor job and queue status. Figure A-9 shows an example orchestration seqence for running an application program within a batch computing job.

Related Capabilities: Extends

- *Parameter Configuration*

Interactions: Command

- `CancelJob()`

  Purpose: Cancel a batch job. Only valid for jobs with `jobStatus` of "JOB_QUEUED" or "JOB_RUNNING".

  Command Data: `dataName` (Data Type) : Data description.

  - `jobId` (`String`) : The job identifier for the batch job.

- `HoldJob()`

  Purpose: Place a queue hold on a batch job. Only valid for jobs in the "JOB_QUEUED" `jobStatus`.

  Command Data: `dataName` (Data Type) : Data description.

  - `jobId` (`String`) : The job identifier for the batch job.

- `ReleaseJob()`

  Purpose: Release a queue hold on a batch job. Only valid for jobs in the "JOB_HELD" `jobStatus`.

  Command Data: `dataName` (Data Type) : Data description.

  - `jobId` (`String`) : The job identifier for the batch job.

Interactions: Request-Reply

- `GetJobInfo()`

  Purpose: Request detailed job information. The format of the returned information is unspecified and may vary across batch job queueing systems.

  Request Data: `dataName` (Data Type) : Data description.

  - `jobId` (`String`) : The job identifier for the batch job.

  Reply Data: `dataName` (Data Type) : Data description.

  - `jobInfo` (`String`) : The detailed job information for the given `jobId`.
  - `errorMsg` (`String`) : (Optional) An error message describing why the requested job information could not be returned for the given `jobId`.

- `GetJobStatus()`

  Purpose: Request the current job status.

  Request Data: `dataName` (Data Type) : Data description.

  - `jobId` (`String`) : The job identifier for the batch job.

  Reply Data: `dataName` (Data Type) : Data description.

  - `jobStatus` (`String`) : The job status information for the given `jobId` (e.g., "JOB_QUEUED", "JOB_HELD", "JOB_RUNNING", or "JOB_COMPLETED").
  - `errorMsg` (`String`) : (Optional) An error message describing why the requested job status could not be returned for the given `jobId`.

- `GetQueueInfo()`

  Purpose: Request detailed queue information, such as job resource limits. The format of the returned information is unspecified and may vary across batch job queueing systems.

  Request Data: `dataName` (Data Type) : Data description.

  - `queueId` (`String`) : The identifier for the batch queue.

  Reply Data: `dataName` (Data Type) : Data description.

  - `queueInfo` (`String`) : The detailed queue information for the given `queueId`.
  - `errorMsg` (`String`) : (Optional) An error message describing why the requested queue information could not be returned for the given `queueId`.

- `GetQueueStatus()`

  Purpose: Request the current queue status.

  Request Data: `dataName` (Data Type) : Data description.

  - `queueId` (`String`) : The identifier for the batch queue.

  Reply Data: `dataName` (Data Type) : Data description.

  - `queueStatus` (`String`) : The queue status information for the given `queueId` (e.g., "QUEUE_ENABLED", "QUEUE_DISABLED", or "QUEUE_PAUSED").
  - `errorMsg` (`String`) : (Optional) An error message describing why the requested queue status could not be returned for the given `queueId`.

- `ListQueues()`

  Purpose: Request the list of batch queues.

  Request Data: None

  Reply Data: `dataName` (Data Type) : Data description.

  - `queueIdList` (`StringList`) : A list of batch queue identifiers (i.e., `queueId`).

- SubmitJob()

  Purpose: Submit a job script to a particular batch queue.

  Request Data: dataName (Data Type) : Data description.

  - queueId (String) : The identifier for the batch queue.
  - jobScript (Filepath) : The file name of the job script.
  - submitArgs (List< KeyVal<String> >) : A list of key-value parameters for use as arguments to the batch job queueing system submission command. The parameter names may vary across batch job queueing systems, and may be queried via the *Parameter Configuration* capability.

  Reply Data: dataName (Data Type) : Data description.

  - jobId (String) : The identifier for the submitted job.
  - errorMsg (String) : (Optional) An error message describing why the job could not be submitted to the given queueId.

Interactions: Asynchronous Status

- JobStatusChange

  Purpose: Notification of job status changes.

  Event Data: dataName (Data Type) : Data description.

  - jobId (String) : The job identifier for the batch job.
  - jobStatus (String) : The job status information for the given jobId (e.g., "JOB_QUEUED", "JOB_HELD", "JOB_RUNNING", or "JOB_COMPLETED").

- QueueStatusChange

  Purpose: Notification of queue status changes.

  Event Data: dataName (Data Type) : Data description.

  - queueId (String) : The identifier for the batch queue.
  - queueStatus (String) : The queue status information for the given queueId (e.g., "QUEUE_ENABLED", "QUEUE_DISABLED", or "QUEUE_PAUSED").

3.2.4.4 Capability: *Compute Queue Reservation*

Description: Interact with batch queueing systems to reserve dedicated compute system access, either at a specified time or by some deadline.

Related Capabilities: Extends

- *Compute Queue*

Interactions: Command

- `CancelReservation()`

  Purpose: Cancel a batch queue reservation. Only valid for reservations with `reservationStatus` of "RESERVATION_GRANTED".

  Command Data: `dataName` (Data Type) : Data description.

  – `reservationId` (`String`) : The batch queue reservation identifier.

Interactions: Request-Reply

- `CreateReservation()`

  Purpose: Create a batch queue reservation at a specific time.

  Request Data: `dataName` (Data Type) : Data description.

  – `queueId` (`String`) : The identifier for the batch queue.

  – `hostCount` (`Integer`) : The number of compute hosts to reserve.

  – `wallTime` (`TimeDuration`) : The requested walltime for the reservation.

  – `startTime` (`Timestamp`) : The requested start time for the reservation.

  Reply Data: `dataName` (Data Type) : Data description.

  – `reservationId` (`String`) : The batch queue reservation identifier.

  – `errorMsg` (`String`) : (Optional) An error message describing why the requested batch queue reservation could not be granted.

- `CreateDeadlineReservation()`

  Purpose: Create a batch queue reservation that should end no later than a specified time deadline.

  Request Data: `dataName` (Data Type) : Data description.

  – `queueId` (`String`) : The identifier for the batch queue.

  – `hostCount` (`Integer`) : The number of compute hosts to reserve.

  – `wallTime` (`TimeDuration`) : The requested walltime for the reservation.

- deadlineTime (Timestamp) : The requested deadline time for the end of the reservation.

Reply Data: dataName (Data Type) : Data description.

- reservationId (String) : The batch queue reservation identifier.
- errorMsg (String) : (Optional) An error message describing why the requested batch queue reservation could not be granted.

- GetReservationStatus()

  Purpose: Request the current reservation status.

  Request Data: dataName (Data Type) : Data description.

  - reservationId (String) : The batch queue reservation identifier.

  Reply Data: dataName (Data Type) : Data description.

  - reservationStatus (String) : The batch queue reservation status information for the given reservationId (e.g., "RESERVATION_GRANTED", "RESERVATION_ACTIVE", or "RESERVATION_ENDED").
  - errorMsg (String) : (Optional) An error message describing why the requested batch queue reservation status could not be returned for the given reservationId.

- SubmitJobToReservation()

  Purpose: Submit a job script to a particular batch queue reservation.

  Request Data: dataName (Data Type) : Data description.

  - reservationId (String) : The batch queue reservation identifier.
  - jobScript (Filepath) : The file name of the job script.
  - submitArgs (List< KeyVal<String> >) : A list of key-value parameters for use as arguments to the batch job queueing system submission command. The parameter names may vary across batch job queueing systems.

  Reply Data: dataName (Data Type) : Data description.

  - jobId (String) : The identifier for the submitted job.
  - errorMsg (String) : (Optional) An error message describing why the job could not be submitted to the given reservationId.

Interactions: Asynchronous Status

- ReservationStatusChange

  Purpose: Notification of batch queue reservation status changes.

  Event Data: dataName (Data Type) : Data description.

  - reservationId (String) : The batch queue reservation identifier.

- `reservationStatus` (`String`) : The batch queue reservation status information for the given `reservationId` (e.g., "RESERVATION_GRANTED", "RESERVATION_ACTIVE", or "RESERVATION_ENDED").

3.2.4.5 Capability: *Application Execution*

Description: Executes a packaged application on a computing system. A packaged application is one that has been previously deployed and configured for use on the target computing system. Packaged applications should also integrate with INTERSECT Data and Information Management services to store and publish any data products or streams.

Related Capabilities: Extends

- *Parameter Configuration*

Related Capabilities: Requires

- *Compute Allocation* **OR** *Compute Queue*

Interactions: Command

- `RunApplication()`

  Purpose: Run the application using the given `runParameters`.

  Command Data: `dataName` (Data Type) : Data description.

  - `runId` (`String`) : The client's identifier for the application run.
  - `runParameters` (`List< KeyVal<String> >`) : A list of key-value parameters to control execution behavior. The parameter names may vary across applications and may be queried via the *Parameter Configuration* capability.

Interactions: Request-Reply

- `GetApplicationExecutionStatus()`

  Purpose: Request the current execution status for the given `runId`..

  Request Data: `dataName` (Data Type) : Data description.

  - `runId` (`String`) : The client's identifier for the application run.

  Reply Data: `dataName` (Data Type) : Data description.

  - `execStatus` (`String`) : The execution status information for the given `runId` (e.g., "EXECUTION_PENDING", "EXECUTION_IN_PROGRESS", "EXECUTION_COMPLETED", or "EXECUTION_FAILED").
  - `failureMsg` (`String`) : (Optional) When the application execution has failed, an error message describing any available details of the failure.
  - `errorMsg` (`String`) : (Optional) An error message describing why the requested execution status could not be returned for the given `runId`.

- `GetApplicationDescription()`

Purpose: Request a textual description of the application. The description should include information regarding the intended purpose of the application, how it was packaged (e.g., associated source code and versions), and how it was configured for the target system.

Reply Data: `dataName` (Data Type) : Data description.

- `appDescription` (`String`) : The application description text.

Interactions: Asynchronous Status

- `ApplicationExecutionStatusChange`

  Purpose: Notification of application execution status changes.

  Event Data: `dataName` (Data Type) : Data description.

  - `runId` (`String`) : The client's identifier for the application run.
  - `execStatus` (`String`) : The execution status information for the given `runId` (e.g., "EXECUTION_PENDING", "EXECUTION_IN_PROGRESS", "EXECUTION_COMPLETED", or "EXECUTION_FAILED").

3.2.4.6 Capability: *Container Execution*

Description: Executes a containerized application on a computing system.

Related Capabilities: Extends

- *Parameter Configuration*

Related Capabilities: Requires

- *Compute Allocation* **OR** *Compute Queue*

Interactions: Command

- BuildContainerImage()

  Purpose: Build and store for later use a container image with given imageName using the given image and build parameters. If the image is built successfully, will trigger the ContainerImageAvailable event.

  Command Data: dataName (Data Type) : Data description.

  - imageName (String) : The name to use for the container image.
  - imageSpec (String) : The build context for the container image (e.g., a local file path or URL).
  - imageFile (String) : (Optional) The file name relative to the build context that contains the image build commands.
  - buildParameters (List< KeyVal<String> >) : (Optional) A list of key-value parameters to control build behavior. The parameter names may vary across container environments and may be queried via the *Parameter Configuration* capability.

- RemoveContainerImage()

  Purpose: Remove a previously built container image with given imageName. If the image is successfully removed, will trigger the ContainerImageRemoval event.

  Command Data: dataName (Data Type) : Data description.

  - imageName (String) : The name to use for the container image.

Interactions: Request-Reply

- RunContainer()

  Purpose: Build and run the container application using the given image, build, and run parameters. The image is not stored for later use. The reply should be delayed until a build or run error is encountered, or until the container has been successfully launched.

  Request Data: dataName (Data Type) : Data description.

  - runId (String) : The client's identifier for the container application's execution.

- imageSpec (String) : The build context for the container image (e.g., a local file path or URL).
- imageFile (String) : (Optional) The file name relative to the build context that contains the image build commands.
- buildParameters (List< KeyVal<String> >) : (Optional) A list of key-value parameters to control build behavior. The parameter names may vary across container environments and may be queried via the *Parameter Configuration* capability.
- runParameters (List< KeyVal<String> >) : (Optional) A list of key-value parameters to control container execution behavior. The parameter names may vary across container environments and may be queried via the *Parameter Configuration* capability.

Reply Data: dataName (Data Type) : Data description.

- buildErrorMsg (String) : (Optional) An error message describing why the container image could not be built.
- runErrorMsg (String) : (Optional) An error message describing why the container application could not be launched.

• RunContainerFromImage()

Purpose: Run the container application using the given imageName and run parameters.

Request Data: dataName (Data Type) : Data description.

- runId (String) : The client's identifier for the container application's execution.
- imageName (String) : The name of the container image to use.
- runParameters (List< KeyVal<String> >) : (Optional) A list of key-value parameters to control container execution behavior. The parameter names may vary across container environments and may be queried via the *Parameter Configuration* capability.

Reply Data: dataName (Data Type) : Data description.

- errorMsg (String) : (Optional) An error message describing why the container application could not be launched.

• GetContainerExecutionStatus()

Purpose: Request the current execution status for the given runId.

Request Data: dataName (Data Type) : Data description.

- runId (String) : The client's identifier for the container application's execution.

Reply Data: dataName (Data Type) : Data description.

- `execStatus` (`String`) : The execution status information for the given `runId` (e.g., "EXECUTION_IN_PROGRESS", "EXECUTION_COMPLETED", or "EXECUTION_FAILED").
- `failureMsg` (`String`) : (Optional) When the container execution has failed, an error message describing any available details of the failure.
- `errorMsg` (`String`) : (Optional) An error message describing why the requested execution status could not be returned for the given `runId`.

Interactions: Asynchronous Status

- `ContainerExecutionStatusChange`

  Purpose: Notification of container application execution status changes.

  Event Data: `dataName` (Data Type) : Data description.

  - `runId` (`String`) : The client's identifier for the container application's execution.
  - `execStatus` (`String`) : The execution status information for the given `runId` (e.g., "EXECUTION_IN_PROGRESS", "EXECUTION_COMPLETED", or "EXECUTION_FAILED").

- `ContainerImageAvailable`

  Purpose: Notification of container image availability.

  Event Data: `dataName` (Data Type) : Data description.

  - `imageName` (`String`) : The name of the container build image.

- `ContainerImageRemoval`

  Purpose: Notification of container image removal.

  Event Data: `dataName` (Data Type) : Data description.

  - `imageName` (`String`) : The name of the container build image.

3.2.4.7 Capability: *Host Command Execution*

Description: Executes an arbitrary host command on a computing system.

Related Capabilities: Requires

- *Compute Allocation* **OR** *Compute Queue*

Interactions: Request-Reply

- RunHostCommand()

  Purpose: Run a host system command with optional environment settings and return the exit code. No command output is captured.

  Request Data: dataName (Data Type) : Data description.

  - command (String) : The full command to run.
  - runEnvironment (List< KeyVal<String> >) : (Optional) A list of key-value environment settings to use during command execution.

  Reply Data: dataName (Data Type) : Data description.

  - exitCode (Integer) : The exit status of the command.
  - errorMsg (String) : (Optional) An error message describing why the system command could not be run.

- RunHostCommandWithOutputFiles()

  Purpose: Run a host system command with optional environment settings and return the exit code. Command output is captured in the given local file(s).

  Request Data: dataName (Data Type) : Data description.

  - command (String) : The full command to run.
  - stdoutFile (Filepath) : The absolute path to a local file that should be used to capture the stdout of the command.
  - stderrFile (Filepath) : The absolute path to a local file that should be used to capture the stderr of the command. If both stdout and stderr should be combined in one file, then stdoutFile and stderrFile should both refer to the same file.
  - runEnvironment (List< KeyVal<String> >) : (Optional) A list of key-value environment settings to use during command execution.

  Reply Data: dataName (Data Type) : Data description.

  - exitCode (Integer) : The exit status of the command.
  - errorMsg (String) : (Optional) An error message describing why the system command could not be run.

3.3 EXPERIMENT-SPECIFIC MICROSERVICE CAPABILITIES

In this section, we classify INTERSECT experiment-specific microservices into three groups according to their purpose: (1) Experiment Control, (2) Management of Experiment Data, and (3) Experiment Design.

3.3.1 Experiment Control Microservices

Services for experiment control enable the execution of planned tests using scientific resources. Typical activities include validating test configurations, configuring the resource(s) to match the desired test configurations, and initiating and monitoring the execution of tests.

3.3.1.1 Instrument Adapters

Instrument adapters provide command and control and state inspection functionality for scientific instruments (e.g., electron microscopes, chemical reactors, neutron detectors, robots, etc.). These adapters typically must integrate with instrument control software such as ROS [29] or EPICS [4]. The capabilities provided by instrument adapters should abstract away the operational details of controlling instruments and examining their state or data products. Instead, they should offer high-level functionality that in turn orchestrates potentially complex sequences of instrument control software actions. Instrument adapters should also integrate with INTERSECT Data and Information Management services to store and publish any data products or streams.

3.3.1.2 Capability: *Instrument Controller*

Description: Interact with scientific instruments to initiate and control their actions and activities, and to query their operational state or retrieve data products. Instrument actions are intended to correspond to idempotent command or control operations such as moving a physical object to a specific location, and are not expected to generate data products. The completion status for actions should be reported via events where applicable. Instrument activities are intended to correspond to sequences of instrument control software actions that may result in data products.

Related Capabilities: Extends

- *Parameter Configuration*

Interactions: Command

- `PerformAction()`

  Purpose: Perform an instrument action. When the action completes, triggers the `InstrumentActionCompletion` event.

  Command Data: `dataName` (Data Type) : Data description.

  - `actionName` (`String`) : The name of the action to perform.
  - `actionOptions` (`List< KeyVal<String> >`) : (Optional) A list of key-value options associated with the action. The option names may be queried via the *Parameter Configuration* capability.

- `CancelActivity()`

  Purpose: Cancel an ongoing activity. On successful completion, triggers the `InstrumentActivityStatusChange` event with "ACTIVITY_CANCELED" as the status.

  Command Data: `dataName` (Data Type) : Data description.

  - `activityId` (`String`) : The activity identifier.
  - `reason` (`String`) : A description of the reason for canceling the activity.

Interactions: Request-Reply

- `GetActionDescription()`

  Purpose: Request a textual description of the action. The description should include information regarding the intended purpose of the action.

  Request Data: `dataName` (Data Type) : Data description.

  - `actionName` (`String`) : The name of the instrument action.

  Reply Data: `dataName` (Data Type) : Data description.

  - `actionDescription` (`String`) : The action description text.

- GetActivityDescription()

  Purpose: Request a textual description of the activity. The description should include information regarding the intended purpose of the activity and a summary of any data products.

  Request Data: dataName (Data Type) : Data description.

  - activityName (String) : The name of the instrument activity.

  Reply Data: dataName (Data Type) : Data description.

  - activityDescription (String) : The activity description text.

- GetActivityData()

  Purpose: Request a list of the data products generated by the given activityId.

  Request Data: dataName (Data Type) : Data description.

  - activityId (String) : The unique activity identifier.

  Reply Data: dataName (Data Type) : Data description.

  - products (List<UUID>) : A list of UUIDs for the data products generated by the activity.
  - errorMsg (String) : (Optional) An error message describing why the requested data product list could not be returned for the given activityId.

- GetActivityStatus()

  Purpose: Request the current status for the given activityId..

  Request Data: dataName (Data Type) : Data description.

  - activityId (String) : The unique activity identifier.

  Reply Data: dataName (Data Type) : Data description.

  - activityStatus (String) : The status information for the given activityId (e.g., "ACTIVITY_PENDING", "ACTIVITY_IN_PROGRESS", "ACTIVITY_CANCELED", "ACTIVITY_COMPLETED", or "ACTIVITY_FAILED").
  - statusMsg (String) : (Optional) When the activity has been canceled or failed, a message describing any available details.
  - errorMsg (String) : (Optional) An error message describing why the requested activity status could not be returned for the given activityId.

- ListActions()

  Purpose: Request a list of the valid action names.

  Reply Data: dataName (Data Type) : Data description.

- actionNames (List<String>) : The list of valid action names.
- errorMsg (String) : (Optional) An error message describing why the action list could not be returned.

- ListActivities()

  Purpose: Request a list of the valid activity names.

  Reply Data: dataName (Data Type) : Data description.

  - actionNames (List<String>) : The list of valid activity names.
  - errorMsg (String) : (Optional) An error message describing why the activity list could not be returned.

- StartActivity()

  Purpose: Initiate an instrument activity and return a unique identifier. Changes in the activity's status should trigger the InstrumentActivityStatusChange event.

  Request Data: dataName (Data Type) : Data description.

  - activityName (String) : The name of the activity to perform.
  - activityOptions (List< KeyVal<String> >) : (Optional) A list of key-value options associated with the activity. The option names may be queried via the *Parameter Configuration* capability.
  - activityDeadline (Timestamp) : (Optional) A deadline timestamp for completion of the activity. If the activity has not completed by the given deadline, the activity should be canceled and any intermediate data products should be deleted.

  Reply Data: dataName (Data Type) : Data description.

  - activityId (String) : The unique identifier for the activity.
  - errorMsg (String) : (Optional) An error message describing why the requested activity could not be started.

Interactions: Asynchronous Status

- InstrumentActionCompletion

  Purpose: Notification of instrument action completion.

  Event Data: dataName (Data Type) : Data description.

  - actionName (String) : The instrument action name.
  - actionTimeBegin (Timestamp) : The timestamp when the action was initiated.
  - actionTimeEnd (Timestamp) : The timestamp when the action was completed.
  - actionStatus (String) : The instrument action status information (e.g., "ACTION_SUCCESSFUL", or "ACTION_FAILED").

- `failureMsg` (`String`) : (Optional) When the action has failed, a message describing any available details.

- `InstrumentActivityStatusChange`

  Purpose: Notification of instrument activity status changes.

  Event Data: `dataName` (Data Type) : Data description.

  - `activityId` (`String`) : The unique activity identifier.
  - `activityName` (`String`) : The activity name.
  - `activityStatus` (`String`) : The instrument activity status information for the given `activityId` (e.g., "ACTIVITY_PENDING", "ACTIVITY_IN_PROGRESS", "ACTIVITY_CANCELED", "ACTIVITY_COMPLETED", or "ACTIVITY_FAILED").
  - `statusMsg` (`String`) : (Optional) When the activity has been canceled or failed, a message describing any available details.

3.3.2 Experiment Data Microservices

Experiment Data services are focused on acquisition of raw test data from scientific resources, initial analysis and validation of that data, and making experiment results available for use by further activities in a campaign workflow.

3.3.3 Experiment Design Microservices

Services for *Experiment Design* support development and refinement of experiment plans that are used by *Experiment Control* services to execute experiments on scientific resources. Typical activities include creating and validating experiment plans, and refinement of existing plans based on objective-based analysis of prior experimental results or design space exploration criteria.

The *Experiment Steering* science use case design pattern is a form of experiment plan refinement that is limited to updating the test parameters of an existing plan. In contrast, the *Design of Experiments* science use case design pattern involves generating new experiment plans that may involve changes to test parameters, test sequencing, or the component tests of the plan.

4 ORCHESTRATION AND DEPLOYMENT OF INTERSECT MICROSERVICES

This section introduces microservices architecture design patterns for orchestration and deployment.

4.1 MICROSERVICE ORCHESTRATION DESIGN PATTERNS

Orchestration design patterns are concerned with how microservices work together to accomplish the goals of a specific application use case.

4.1.1 Asynchronous Messaging vs. RESTful Services

As previously introduced in Section 2.1, there are two common communication architectures used for microservices: *client-server* and *asynchronous messaging*. Here we describe the relative merits of each communication architecture, followed by example approaches for implementing the common interaction patterns introduced in Section 2.3 using each architecture.

The predominant approach to client-server communication architecture involves RESTful microservices that provide a synchronous request-response model based on representational state transfer over HTTP. Such a 1:1 pattern is suitable for many forms of requests, including control commands and simple information queries. However, the use of synchronous request-response is not advisable for 1:N interactions where one request is sent to many services, due to the serialization of the requests at the client and the resulting impact to completion latency. Furthermore, the use of RESTful services for use cases involving many microservices often leads to tight coupling that reduces reuse and service independence.

Asynchronous messaging supports both 1:1 and 1:N communication patterns using a message broker that delivers messages associated with a specific topic to any interested parties. When the messages represent events, this communication architecture is often referred to as an event-based architecture. Additional benefits of asynchronous messaging include the ability to independently scale clients and servers, support for concurrent providers of the same service, support for transient services, and simplified communication context management due to the use of a central message broker.

Figure 4-1 shows a typical implementation strategy for the *Command Interaction Pattern* using both client-server communication and asynchronous messaging. With client-server communication, sending the `Command` message involves use of an HTTP POST operation, and the message receipt status acknowledgement maps directly to the HTTP response status. With asynchronous messaging, it is assumed the target microservice has previously subscribed to the topic used for receiving commands, and the client uses a PUBLISH operation to issue the `Command`. The receipt status acknowledgement would typically come from a message broker to indicate whether the `Command` has been successfully queued for later delivery, rather than from the target microservice.

Figure 4-2 shows a typical implementation strategy for the *Request-Reply Interaction Pattern* using both client-server communication and asynchronous messaging. With client-server communication, sending the `Request` message takes the form of an HTTP GET operation, and the `Reply` uses the message body of the HTTP response to return the requested information. With asynchronous messaging, it is assumed the target microservice has previously subscribed to the topic used for receiving requests, and the client uses a PUBLISH operation to that topic to issue the `Request`. The `Reply` in asynchronous messaging corresponds to a PUBLISH operation performed by the microservice. The topic used for the reply is often

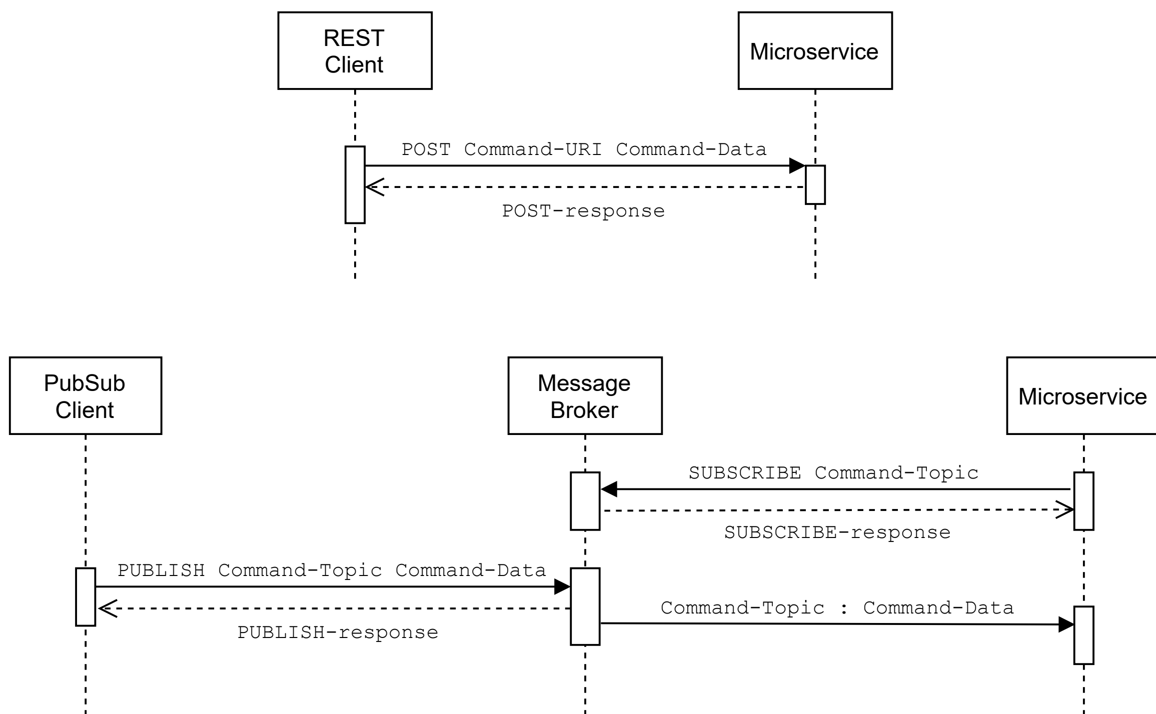

Figure 4-1. Command Interaction Pattern for Client-Server and Asynchronous Messaging

included within the `Request` message contents, or is otherwise made known to the microservice prior to the `Request`.

As shown in Figure 4-3, the *Asynchronous Status or Event Interaction Pattern* is easily implemented using a PUBLISH operation with asynchronous messaging, where interested clients are assumed to have subscribed to the topic to which the `Status` or `Event` is sent. For `Status` messages, the topic should be configured to provide message durability to ensure delivery to at least one subscriber. With client-server communication, however, this pattern is more difficult to support and requires clients to first use a HTTP POST message to register their interest in events with the microservice. Then, when the microservice needs to send an `Event` or `Status`, it iterates to send an HTTP POST message to all interested clients.

4.1.2 Conductor vs. Choreography

Some application use cases involve a sequence of dependent requests to several microservices. The dependencies may be a simple linear sequence where the response from one request is used to make the next request, or more complex such as when multiple responses are combined to form a subsequent request, or when some subset of the requests are actually independent and can be issued concurrently.

Rather than encode the (potentially complex) logic to manage these dependent requests in the client, a common pattern is to offload this logic to another service which provides a simple request-response API to the client. This pattern is known as the Conductor (or Orchestrator) pattern, as the service manages the complexity of many microservices similar to an orchestra conductor managing the performance of many

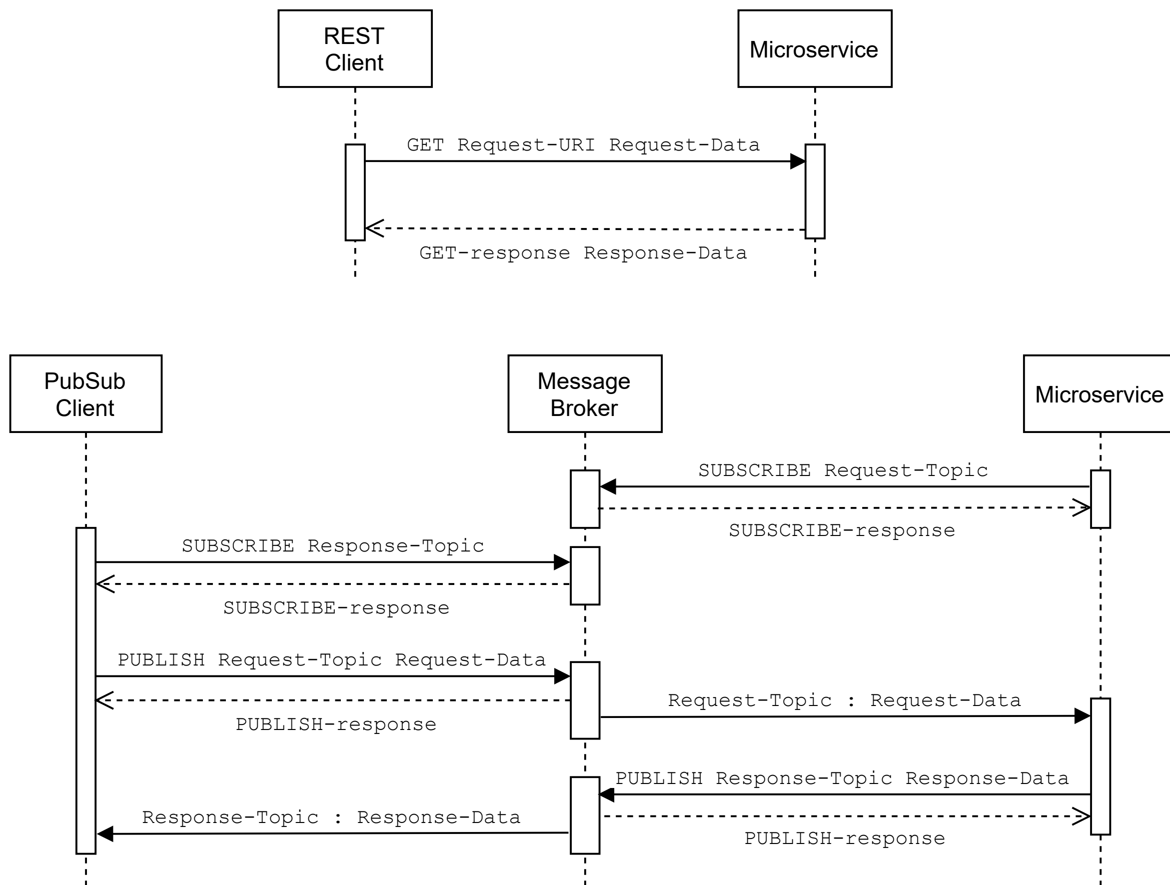

Figure 4-2. Request-Reply Interaction Pattern for Client-Server and Asynchronous Messaging

instrumental sections. For RESTful microservices, the Conductor pattern is appropriate for managing dependent requests. However, in asynchronous messaging architectures the Conductor pattern introduces unnecessary synchronization and latency as responses must be delivered to the conductor, who must then trigger any dependent requests.

For asynchronous messaging, the Choreography pattern is an alternative that translates request dependencies into asynchronous events. Any dependent requests can commence upon receipt of the completion event message for the prior request. The benefits of Choreography for completion latency should be weighed against the need for additional coupling between microservices.

4.2 MICROSERVICE DEPLOYMENT DESIGN PATTERNS

Deployment design patterns focus on approaches that avoid common problems encoutered during software development, operations, and maintenance of microservices.

Software development design patterns for microservices architecture exist to aid in decomposition and composition. For decomposition, the patterns are useful for understanding how to properly abstract

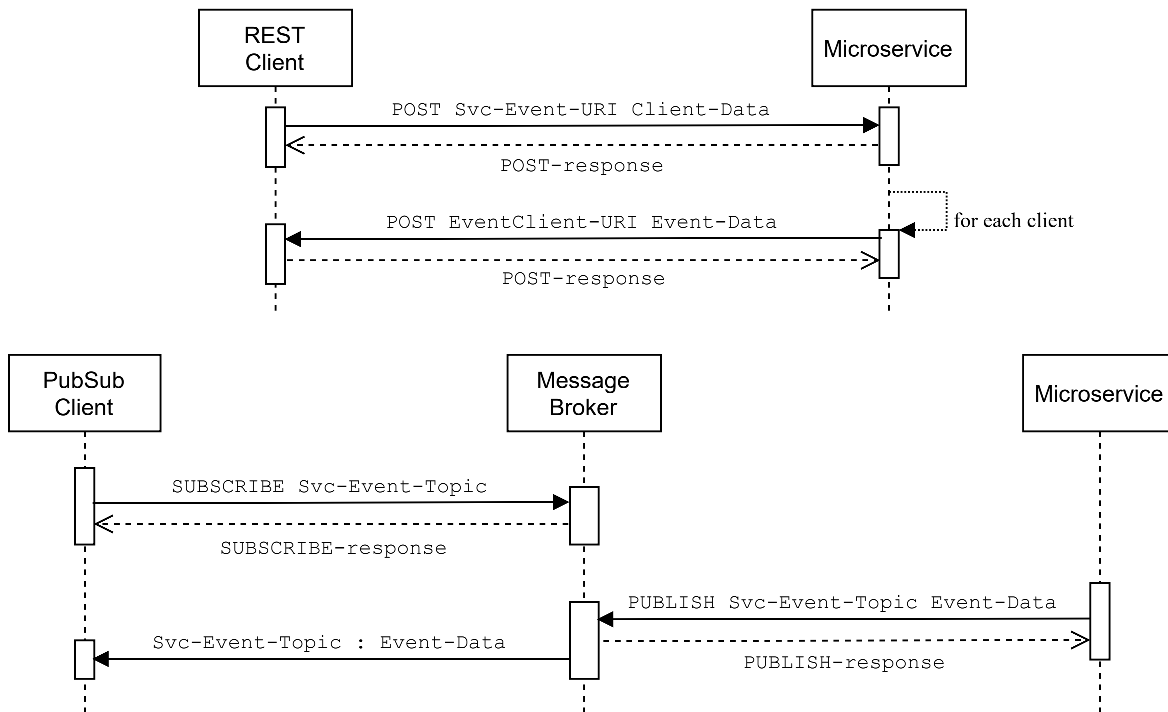

Figure 4-3. Asynchronous Status or Event Interaction Pattern for Client-Server and Asynchronous Messaging

functionality that is common to many microservices, such as the need for logging and monitoring, while still retaining the benefits of service independence and reuse. Composition patterns are useful for addressing problems related to how microservices coordinate within an application use case that involves many services.

4.2.1 Sidecar Pattern

Pattern Name: Sidecar

Pattern Type(s): Software Development, Service Deployment

Problem: Microservices often require common functionality (e.g., configuration management, logging, and monitoring). Developing such functionality within each microservice increases effort, leads to code duplication, and makes both software maintenance and operational deployment more difficult.

Solution: A sidecar service provides common functionality that is independently developed and maintained. The sidecar service is paired with another service known as the primary service. The primary service uses the API of the sidecar to access it capabilities. The sidecar has the same lifecycle as its primary service (i.e., the sidecar is created and destroyed with the primary service).

Related Patterns: Ambassador Proxy Pattern, Service Mesh Pattern

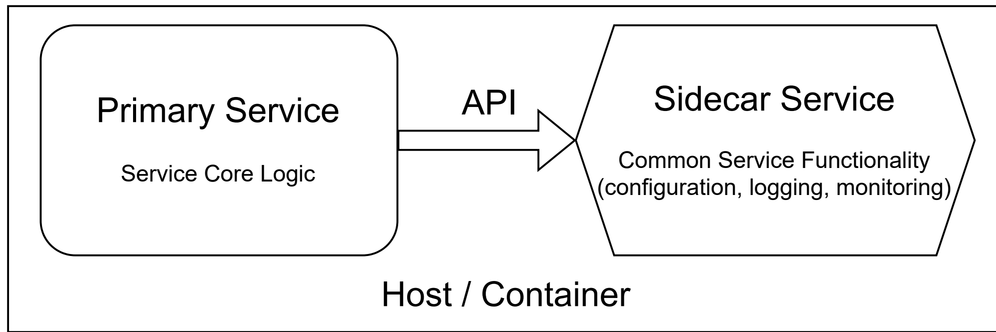

Figure 4-4. Sidecar Pattern

4.2.2 Ambassador Proxy Pattern

Pattern Name: Ambassador Proxy

Pattern Type(s): Software Development, Service Deployment

Problem: Management of communication contexts for remote microservices can be challenging to implement on a per-service basis. Common challenges include service discovery, management of secure communication channels, routing of requests, and communication resiliency.

Solution: An ambassador proxy service encapsulates common features for managing remote connections and requests. The ambassador service is a type of sidecar service that manages remote communication contexts (e.g., service discovery and establishing secure communications) for its primary service. The ambassador service may provide request routing (i.e., selecting a destination service for the request) or communication resilience mechanisms (e.g., timeout and retry).

Related Patterns: Sidecar Pattern, Service Mesh Pattern

4.2.3 Service Mesh Pattern

Pattern Name: Service Mesh

Pattern Type(s): Software Development, Service Deployment, Service Operations

Problem: Distributed applications constructed from many microservices can be difficult to configure and operate.

Solution: A service mesh is an integrated platform that uses the sidecar and ambassador proxy patterns to ease management, improve observability, and decouple core application logic from common microservice functionality for service discovery and communication, logging, request tracing, monitoring, and security. In a service mesh, each service has an associated proxy sidecar through which all communication happens. The proxy manages and captures telemetry for all communication to or from its primary service. Because every service uses a proxy, full observability and tracing of requests is possible.

Related Patterns: Ambassador Proxy Pattern, Sidecar Pattern

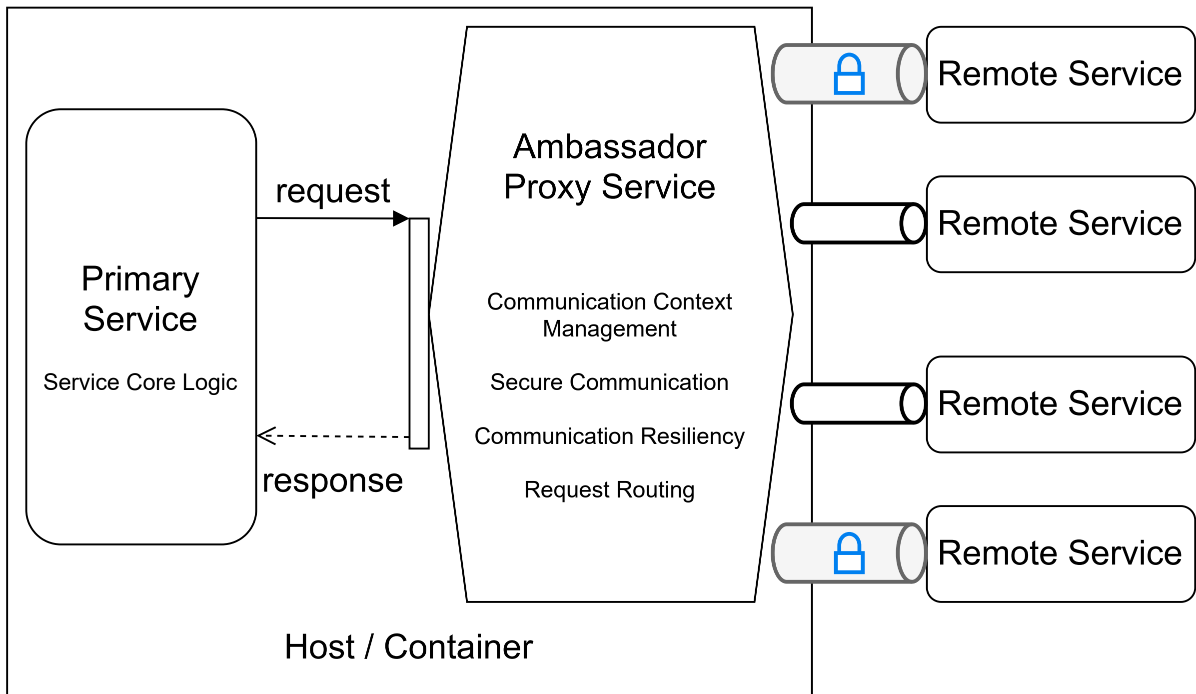

Figure 4-5. Ambassador Proxy Pattern

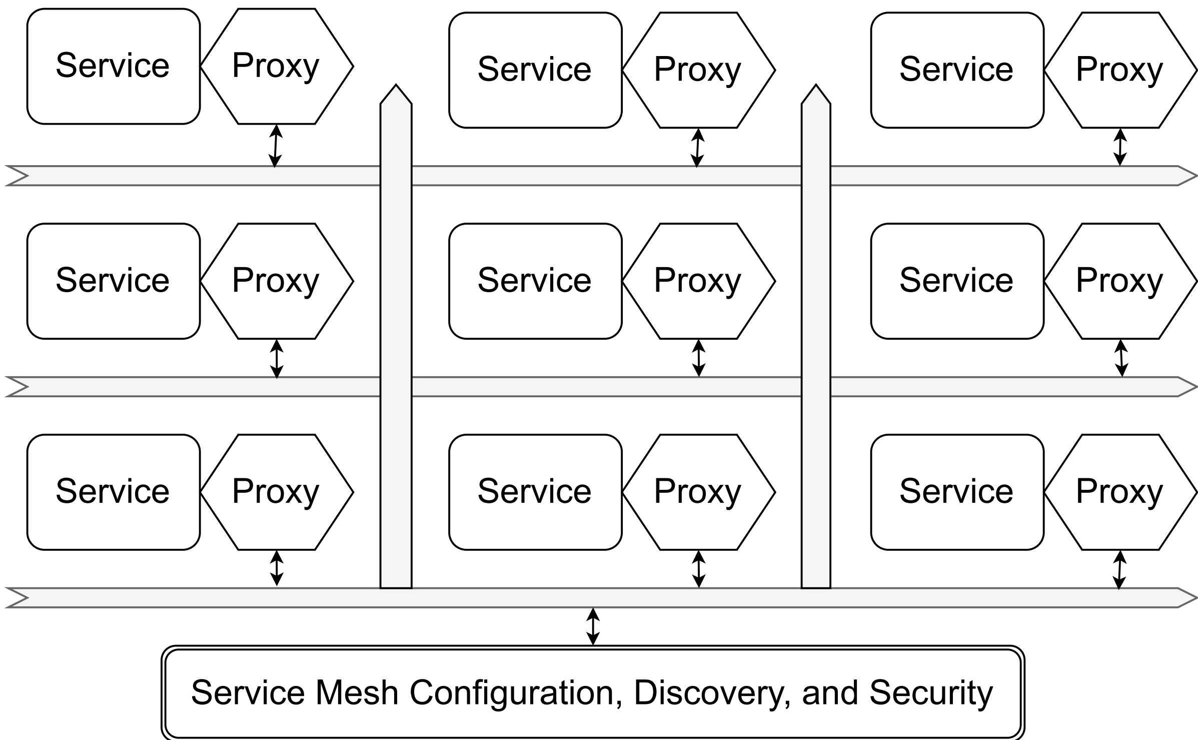

Figure 4-6. Service Mesh Pattern

Bibliography

[1] Semantic Versioning 2.0.0 | Semantic Versioning, 2023. URL https://semver.org/spec/v2.0.0.html.

[2] W. Allcock. GridFTP: Protocol Extensions to FTP for the Grid, 2003. URL http://www.ogf.org/documents/GFD.20.pdf.

[3] SQLite Consortium. SQLite Home Page, 2023. URL https://www.sqlite.org/.

[4] EPICS Controls. EPICS - Experimental Physics and Industrial Control System, 2023. URL https://epics-controls.org/.

[5] Dormando. memcached - a distributed memory object caching system, 2023. URL https://memcached.org/.

[6] Internet Engineering Task Force. JSON Schema: A Media Type for Describing JSON Documents, 2022. URL https://json-schema.org/draft/2020-12/json-schema-core.html.

[7] The Apache Software Foundation. Apache Avro 1.11.0 Documentation, 2021. URL https://avro.apache.org/docs/current/.

[8] The Apache Software Foundation. Apache Cassandra | Apache Cassandra Documentation, 2023. URL https://cassandra.apache.org/.

[9] The Apache Software Foundation. Apache Kafka, 2023. URL https://kafka.apache.org.

[10] Google. Cloud Bigtable: HBase-compatible, NoSQL database | Google Cloud, 2023. URL https://cloud.google.com/bigtable.

[11] Google. Cloud Storage | Google Cloud, 2023. URL https://cloud.google.com/storage.

[12] Google. Personal Cloud Storage & File Sharing Platform - Google, 2023. URL https://www.google.com/drive/.

[13] The PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database, 2023. URL https://www.postgresql.org.

[14] Amazon Web Services Inc. Fast NoSQL Key-Value Database - Amazon DynamoDB - Amazon Web Services, 2023. URL https://aws.amazon.com/dynamodb/.

[15] Amazon Web Services Inc. Amazon Kinesis - Process & Analyze Streaming Data - Amazon Web Services, 2023. URL https://aws.amazon.com/kinesis/.

[16] Amazon Web Services Inc. Fully Managed Graph Database - Amazon Neptune - Amazon Web Services, 2023. URL https://aws.amazon.com/neptune/.

[17] Amazon Web Services Inc. Cloud Object Storage - Amazon S3 - Amazon Web Services, 2023. URL https://aws.amazon.com/s3/.

[18] DropBox Inc. Dropbox.com, 2023. URL https://www.dropbox.com.

[19] MinIO Inc. MinIO | High Performance, Kubernetes Native Object Storage, 2023. URL https://min.io/.

[20] MongoDB Inc. MongoDB: The Developer Data Platform | MongoDB, 2023. URL https://www.mongodb.com.

[21] Neo4j Inc. Neo4j Graph Data Platform | Graph Database Management System, 2023. URL https://neo4j.com.

[22] Argonne National Laboratory. DOE national laboratories' computational facilities – Research workshop report. Technical Report ANL/MCS-TM-388, Argonne National Laboratory, Lemont, IL, USA, February 2020. URL https://publications.anl.gov/anlpubs/2020/02/158604.pdf.

[23] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace, 2005. URL https://www.rfc-editor.org/rfc/rfc4122.html.

[24] Redis Ltd. Redis | The Real-time Data Platform, 2023. URL https://redis.com/.

[25] Microsoft. Azure Blob Storage | Microsoft Azure, 2023. URL https://azure.microsoft.com/en-us/products/storage/blobs/.

[26] Microsoft. Microsoft OneDrive Cloud Storage and File Sharing | Microsoft 365, 2023. URL https://www.microsoft.com/en-us/microsoft-365/onedrive/onedrive-for-business.

[27] Oracle. MySQL, 2023. URL https://www.mysql.com.

[28] AsyncAPI Project. AsyncAPI Specification - Version 2.3.0, 2022. URL https://www.asyncapi.com/docs/specifications/v2.3.0.

[29] Open Robotics. ROS: Home, 2021. URL https://www.ros.org/.

[30] SmartBear Software. OpenAPI specification - version 3.0.3, 2021. URL https://swagger.io/specification/.

[31] Rick Stevens, Valerie Taylor, Jeff Nichols, Arthur Barney Maccabe, Katherine Yelick, and David Brown. AI for science report, March 2020. URL https://www.anl.gov/ai-for-science-report.

[32] University of Chicago and Argonne National Laboratory. Research data management simplified - globus, 2023. URL https://www.globus.org/.

[33] World Wide Web Consortium (W3C). W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures, 2012. URL https://www.w3.org/TR/xmlschema11-1/.

APPENDICES

A INTERSECT INFRASTRUCTURE MICROSERVICE INTERACTION SEQUENCES

In this appendix, we provide example microservice interaction sequence diagrams to document the expected flow of interactions for common scenarios. Figure A-1 shows an example sequence diagram. As shown in the figure, boxes are used to denote INTERSECT systems and services, and individual entity lifelines are shown for each microservice capability within a service. Example messages between services and asynchronous events are shown for each of the three common microservice interaction patterns introduced in Section 2.3.

A-1 Registration of INTERSECT Systems, Services, and Resources

Dynamic registration of INTERSECT systems, resources, and services are crucial activities in the deployment of interconnected scientific ecosystems. Below we provide example orchestration sequences for each of these three registration activities.

Registration of an INTERSECT system is shown in Figure A-2. It is assumed that every INTERSECT system will have an associated management service the coordinates all aspects related to system information management, control of services and subsystems, and status monitoring of associated resources, services, and subsystems. In the figure, this service is called the "System-X Management" service. It provides two key microservice capabilities, the *System Information Catalog* (see 3.2.3.1) and the *System Manager* (see 3.2.3.2). The management service is responsible for registering its parent system with the registrar. The INTERSECT architecture permits a hierarchy of coordinating services providing the *Systems Registrar* capability (see 3.2.3.3). In this figure, we assume each distinct INTERSECT operational domain (e.g., an organization or facility) provides a registrar. The registrar provides a domain-scoped UUID to the system through use of the namespace UUID generation method of the *UUID Generation* capability (see 3.2.1.4).

Registration of resources associated with an INTERSECT system is shown in Figure A-3. The management service is again responsible for registering the resources with the domain registrar to obtain system-scoped UUIDs for each resource. It also needs to update the system information catalog to associated each resource with the parent system and its assigned resource UUID.

Registration of services associated with an INTERSECT system is shown in Figure A-4. The newly deployed instrument adapater service in this figure makes a request to register with the management service, which in turn registers the service with the domain registrar to obtain a system-scoped UUID for the service. The interaction with the the *UUID Generation* capability is elided, but follows the same pattern as that used for assigning system-scoped resource UUIDs in Figure A-3. The managment service then updates the system information catalog to record the capabilities and resources associated with the new service and its assigned service UUID.

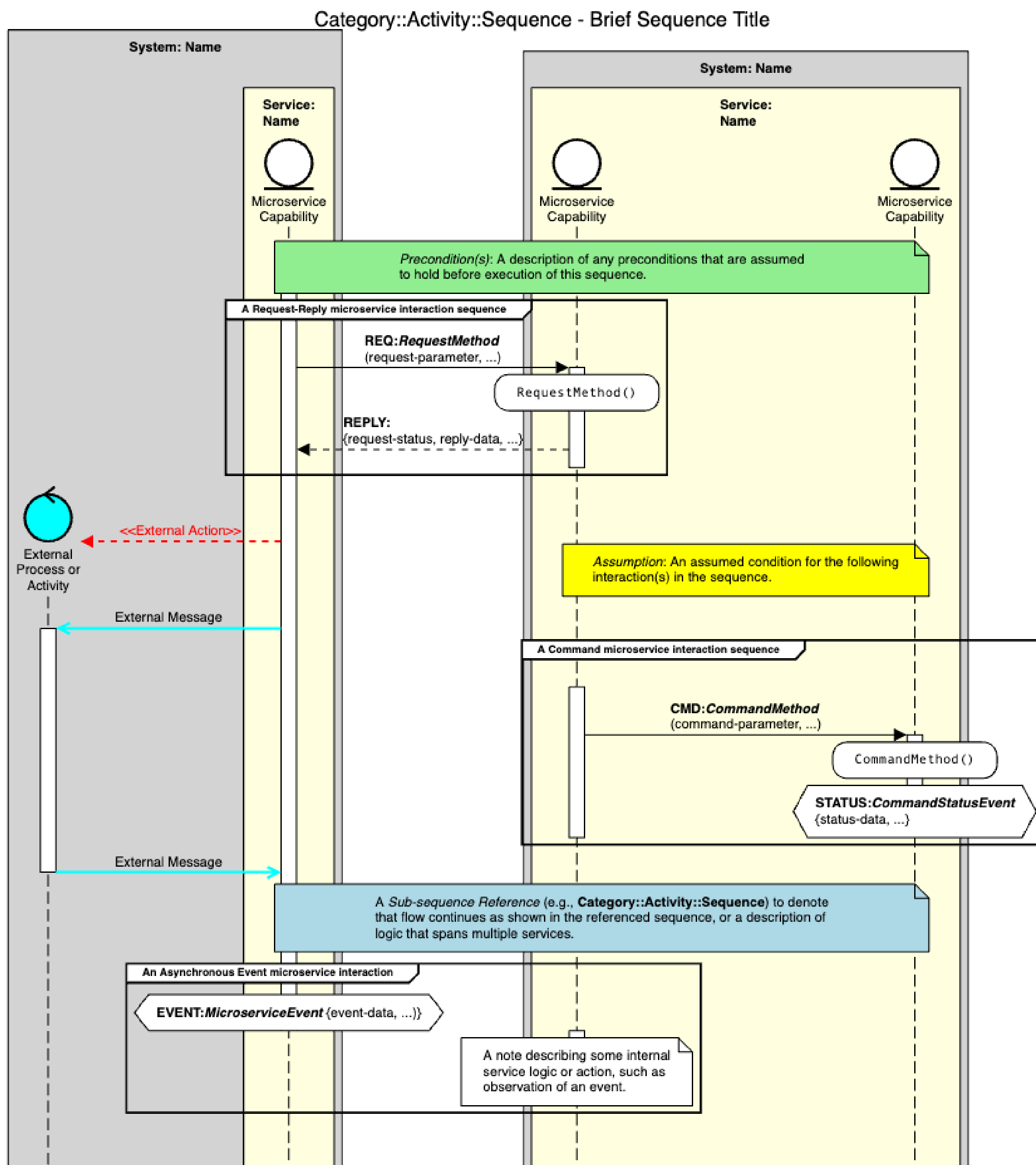

Figure A-1. Example interaction sequence showing common components.

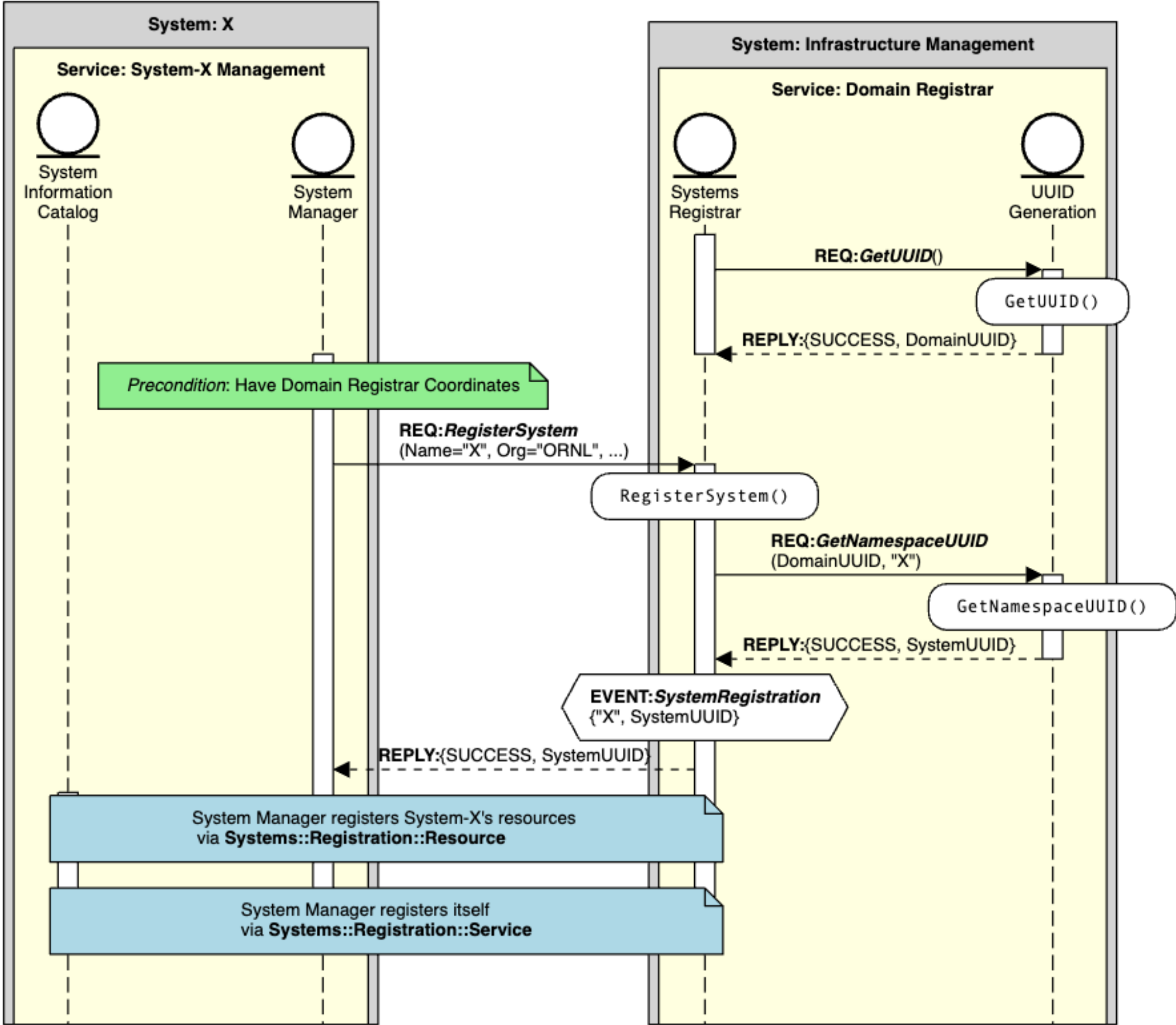

Figure A-2. Sequence for registering an INTERSECT system.

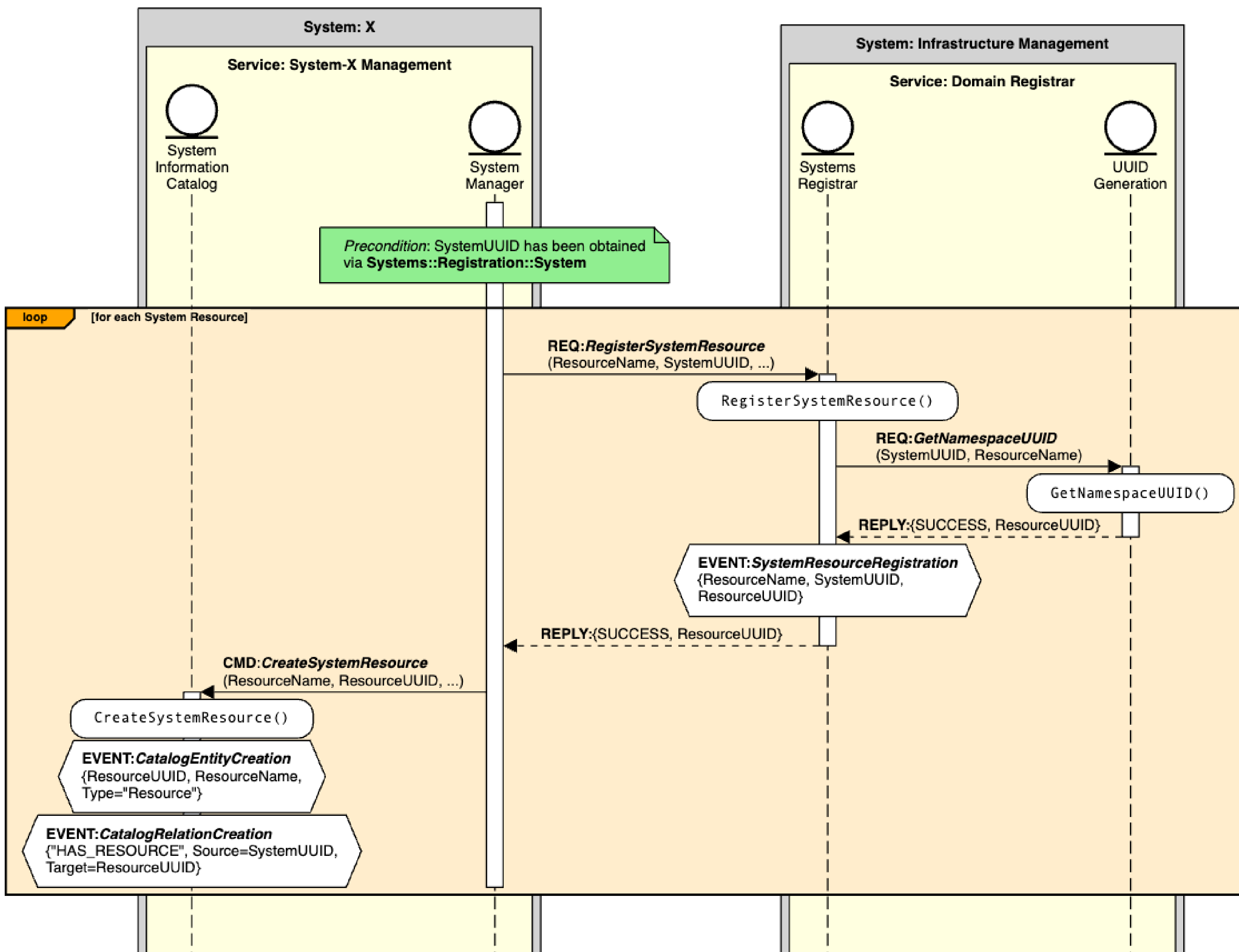

Figure A-3. Sequence for registering INTERSECT system resources.

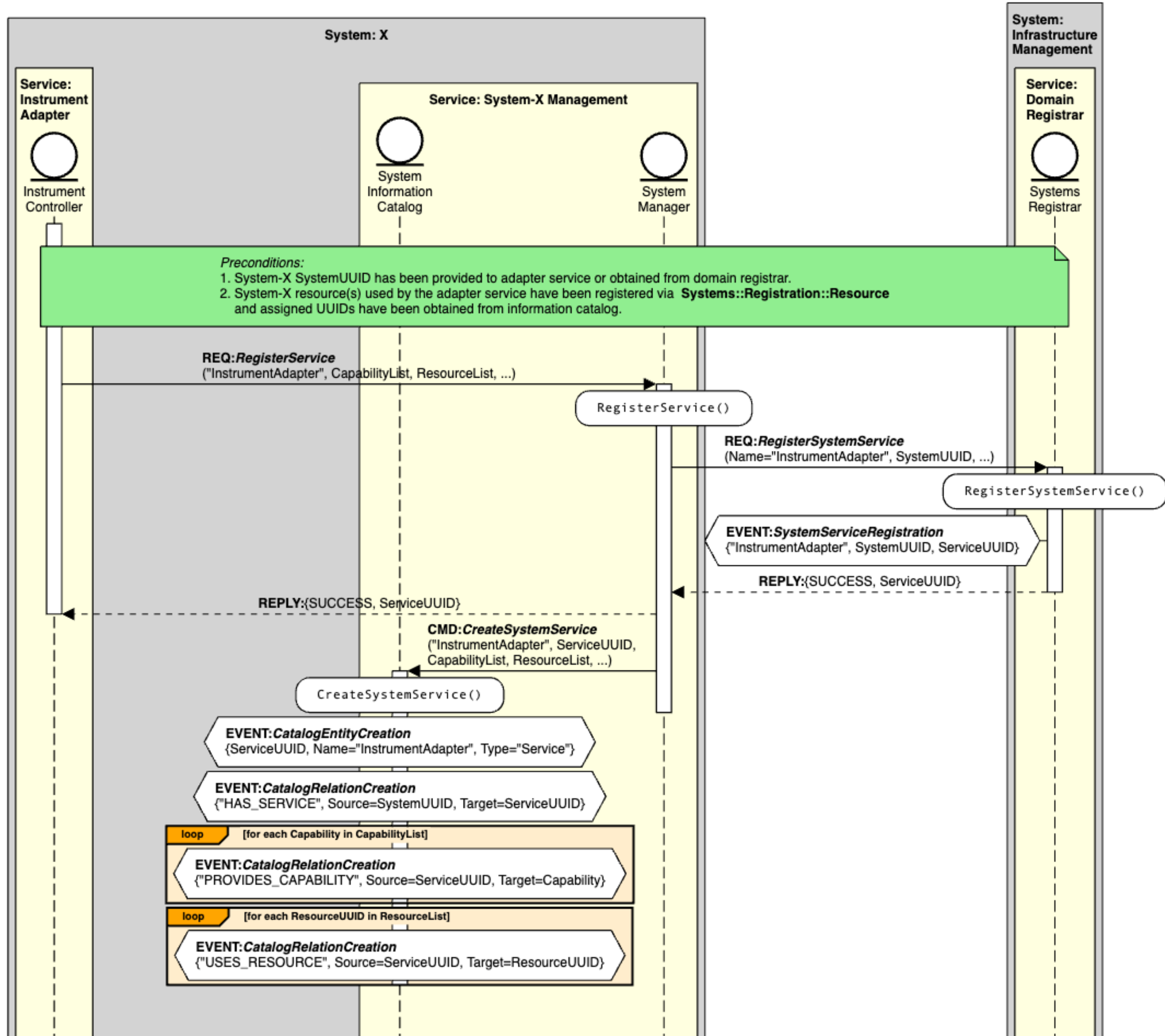

Figure A-4. Sequence for registering an INTERSECT system service.

A-2 Running Applications using On-Demand or Batch Computing Resources

A common scenario for use of computing resources is to deploy an application preconfigured for use in a particular computing environment. The application program and any dependent libraries are built for the target environment and packaged for use in a microservice providing the *Application Execution* capability (see 3.2.4.5). This application deployment microservice provides default settings for the application's run configuration and program arguments. Since the *Application Execution* capability extends the *Parameter Configuration* capability (see 3.2.1.3), the application microservice can also expose any configuration or parameter settings as client tunables.

The ability to run the application via an on-demand allocation or a job script is provided by another microservice that serves as the interface to the target environment's compute resource manager. The compute resource manager microservice provides the *Compute Allocation* capability (see 3.2.4.2), the *Compute Queue* capability (see 3.2.4.3), or both depending on the usage model of the environment.

Figure A-5 shows an example orchestration sequence for running such an application within an on-demand allocation of computing resources. Figure A-6 shows the sub-sequence relating to on-demand allocation of computing resources, while Figure A-7 shows the sub-sequence for running the application program within the allocation.

Figure A-8 shows an example orchestration sequence for running such an application using a batch computing job. Figure A-9 shows the sub-sequence capturing the batch job submission. Figure A-10 shows the sub-sequence to discover available batch computing queues.

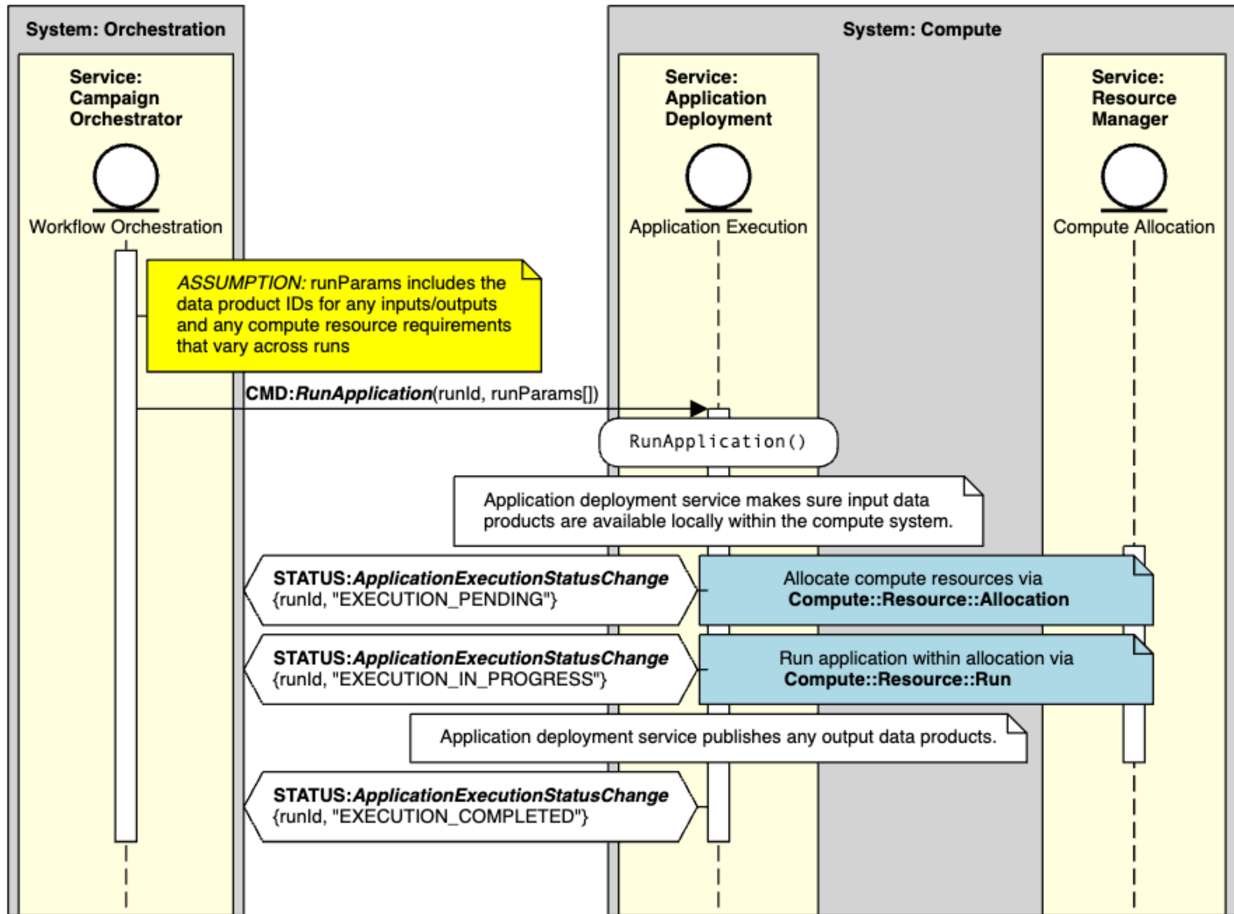

Figure A-5. Sequence for running an application within an on-demand allocation of computing resources.

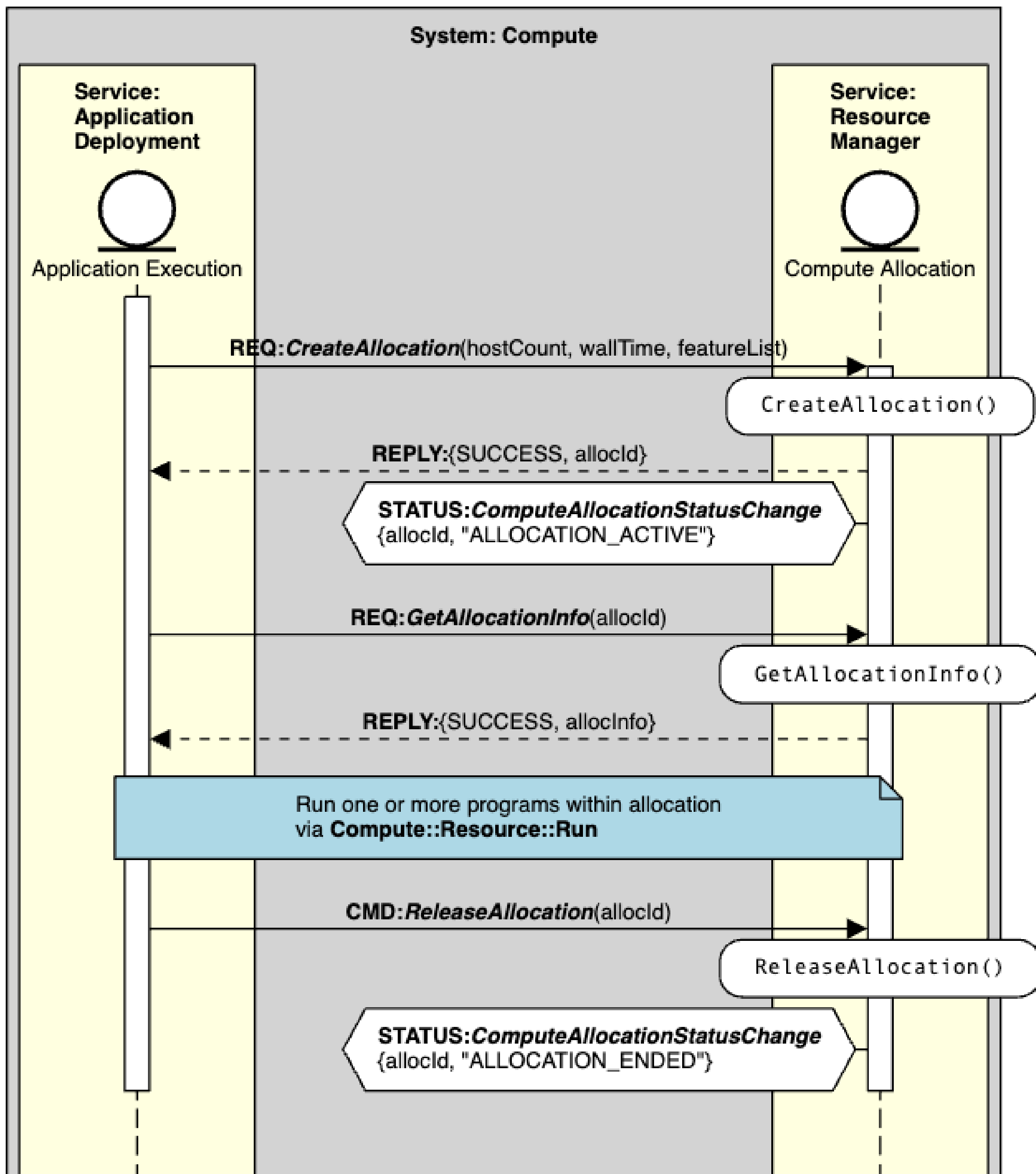

Figure A-6. Sequence for allocating and releasing computing resources.

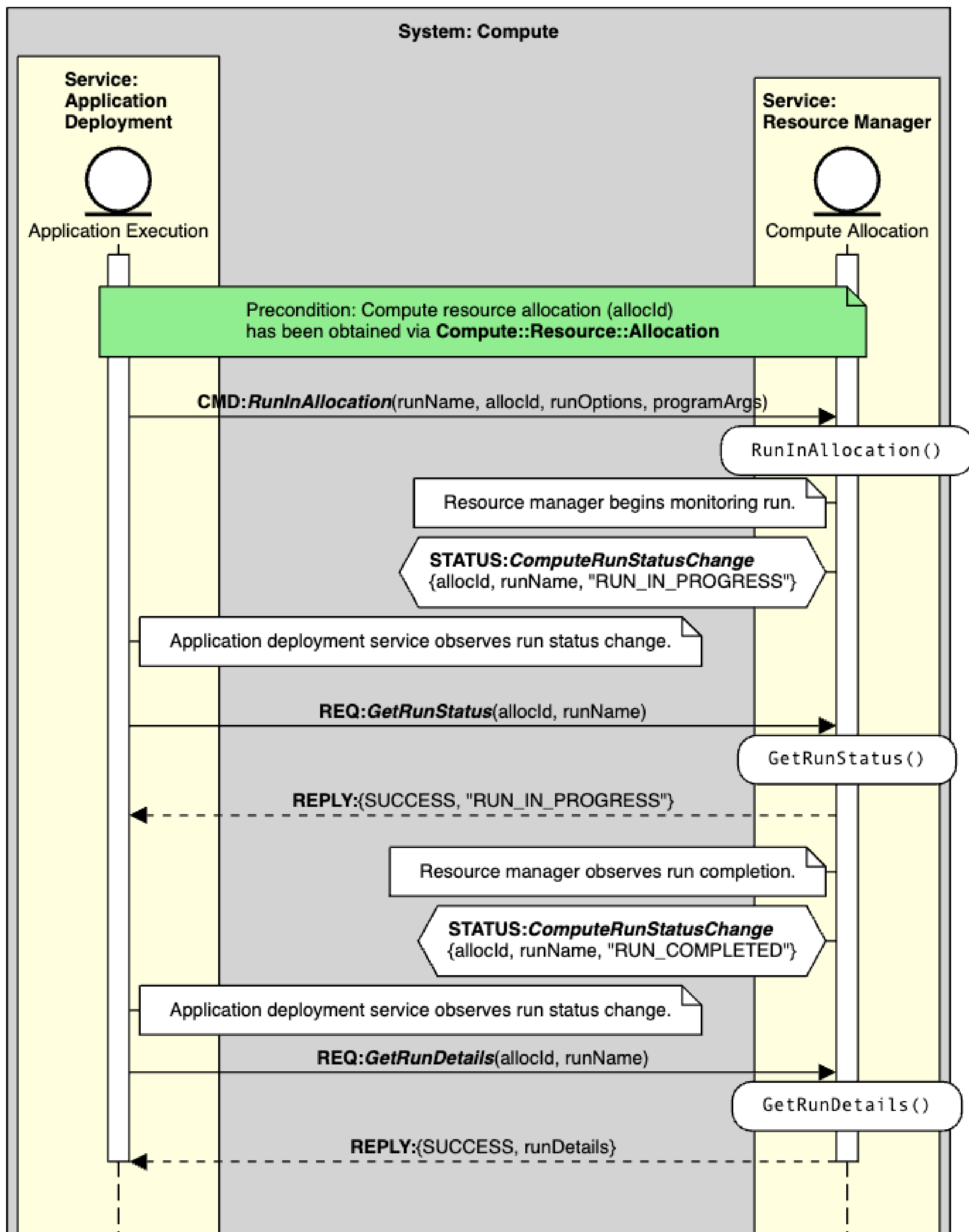

Figure A-7. Sequence for running and monitoring a program within allocated computing resources.

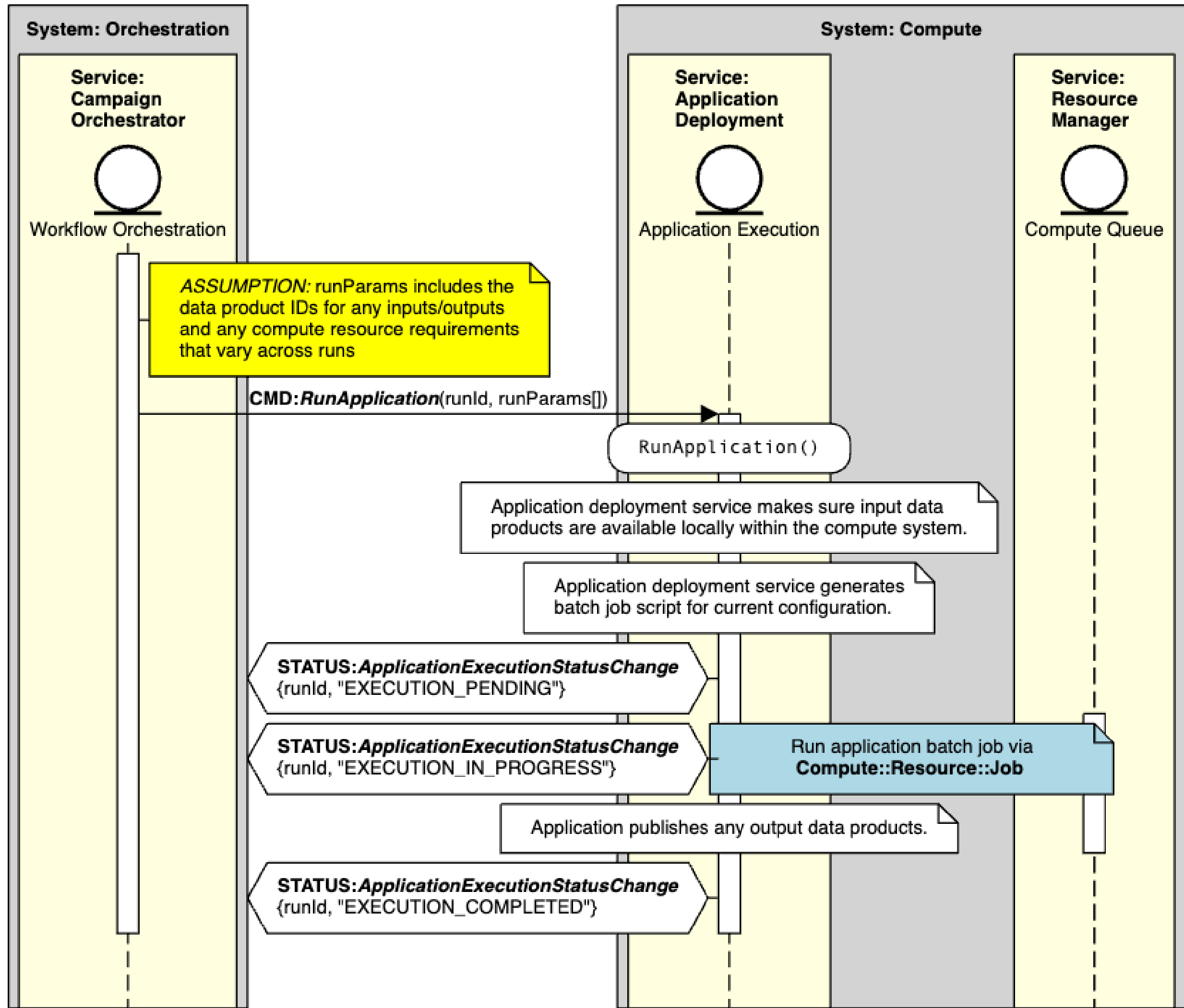

Figure A-8. Sequence for running an application within a batch computing job.

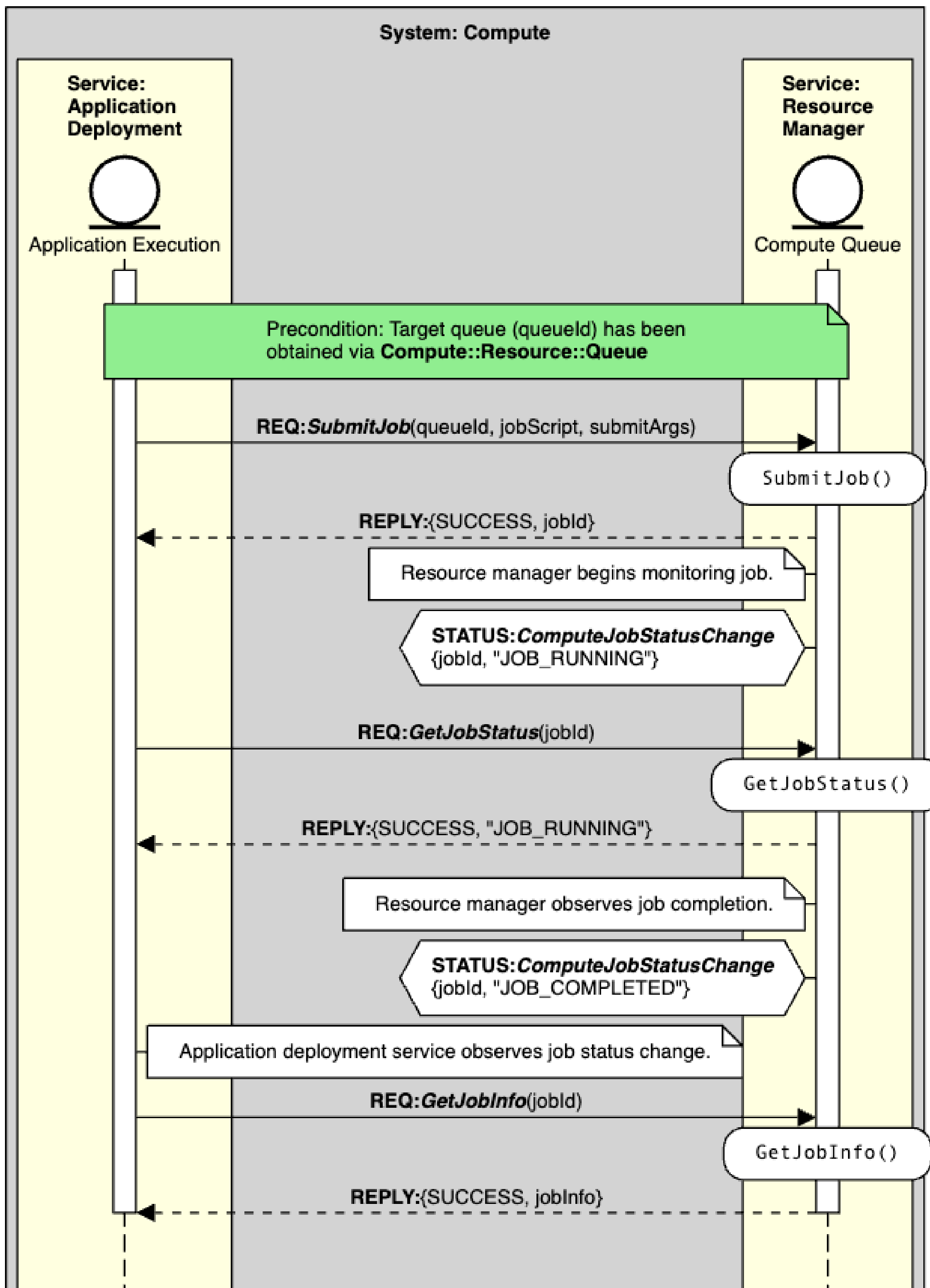

Figure A-9. Sequence for running and monitoring a batch compute job.

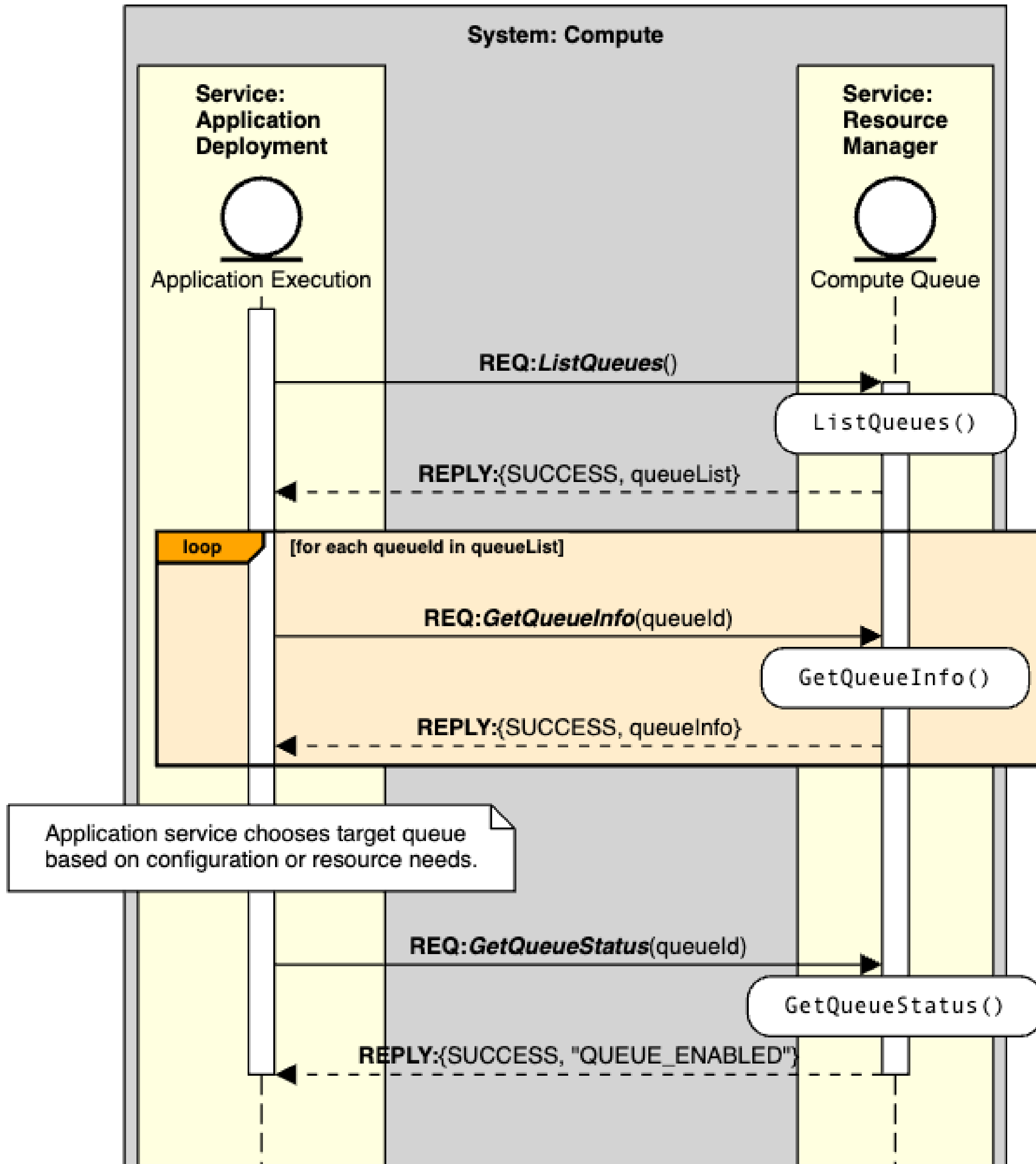

Figure A-10. Sequence for discovery of batch computing queues.

A-3 Data Management for Application Data Products

Explicit management of data products is key to enabling campaign workflow orchestration within INTERSECT ecosystems. Currently, the microservice capabilities for data management include storage (see *Data Storage* 3.2.2.7), publishing (see *Data Catalog* 3.2.2.6), and transfer (see *Data Transfer* 3.2.2.9 and *Data Transfer Endpoint* 3.2.2.10).

An application or service that produces data to be consumed by other applications or services must first store the data item or collection within its system's data management service. Figure A-11 shows an example orchestration sequence for an application to manage its data namespaces. Figure A-12 shows an example orchestration sequence for an application to manage a data collection within its namespace. Figure A-13 shows an example orchestration sequence for an application to manage data items within its namespace.

Once stored, the data product can be published to a data catalog service to alert orchestrators or consumers that the data is available. Figure A-14 shows an example orchestration sequence for a workflow to allocate application data product UUIDs for use by an application to publish its data products to a campaign data catalog.

When the workflow requires the data product(s) to be made available on remote systems, the orchestrator can arrange for the data to be transferred to the data management service of the consumer's system, as shown in Figures A-15, A-16, and A-17.

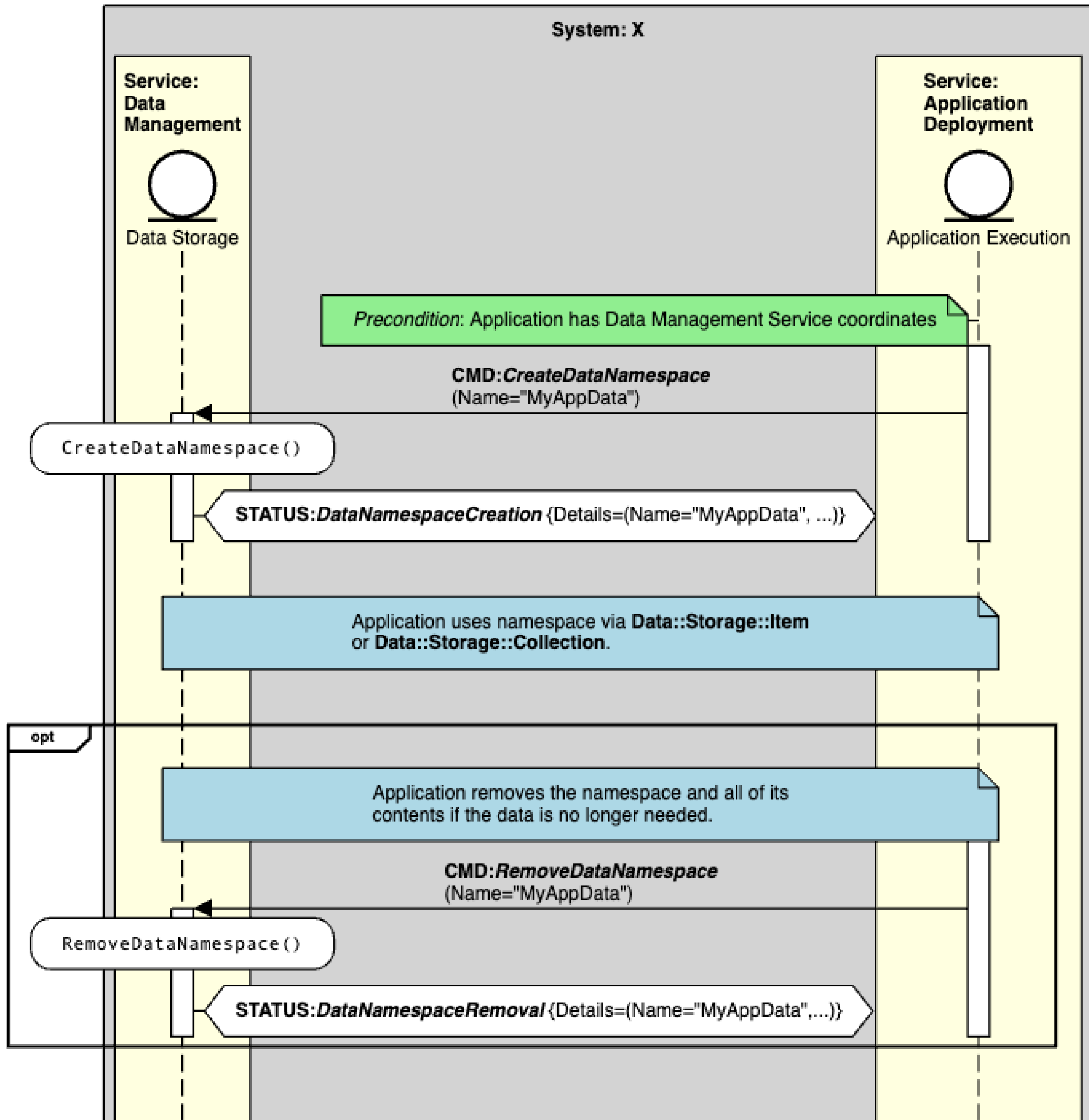

Figure A-11. Sequence for an application to manage a data namespace.

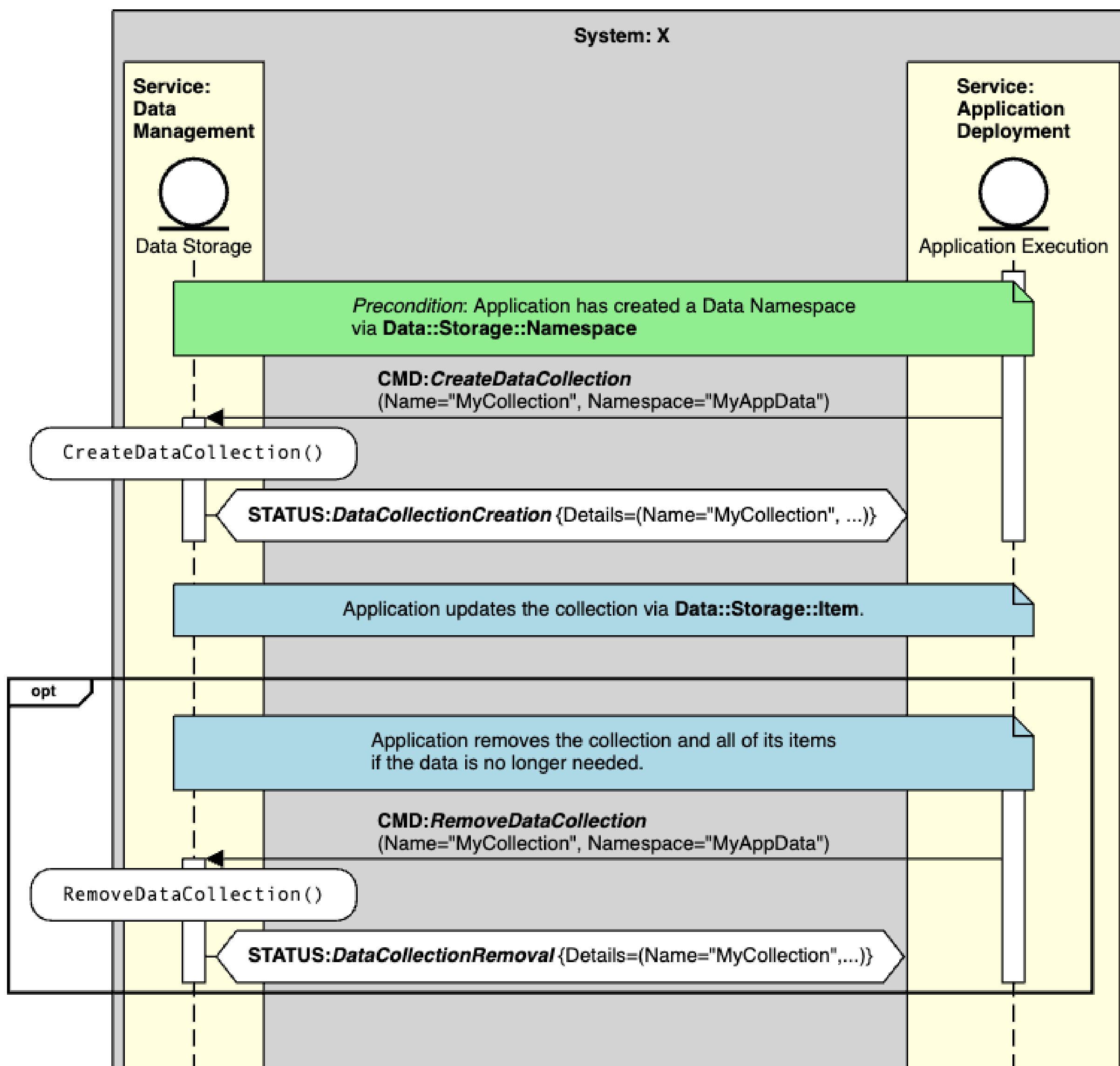

Figure A-12. Sequence for an application to manage a data collection.

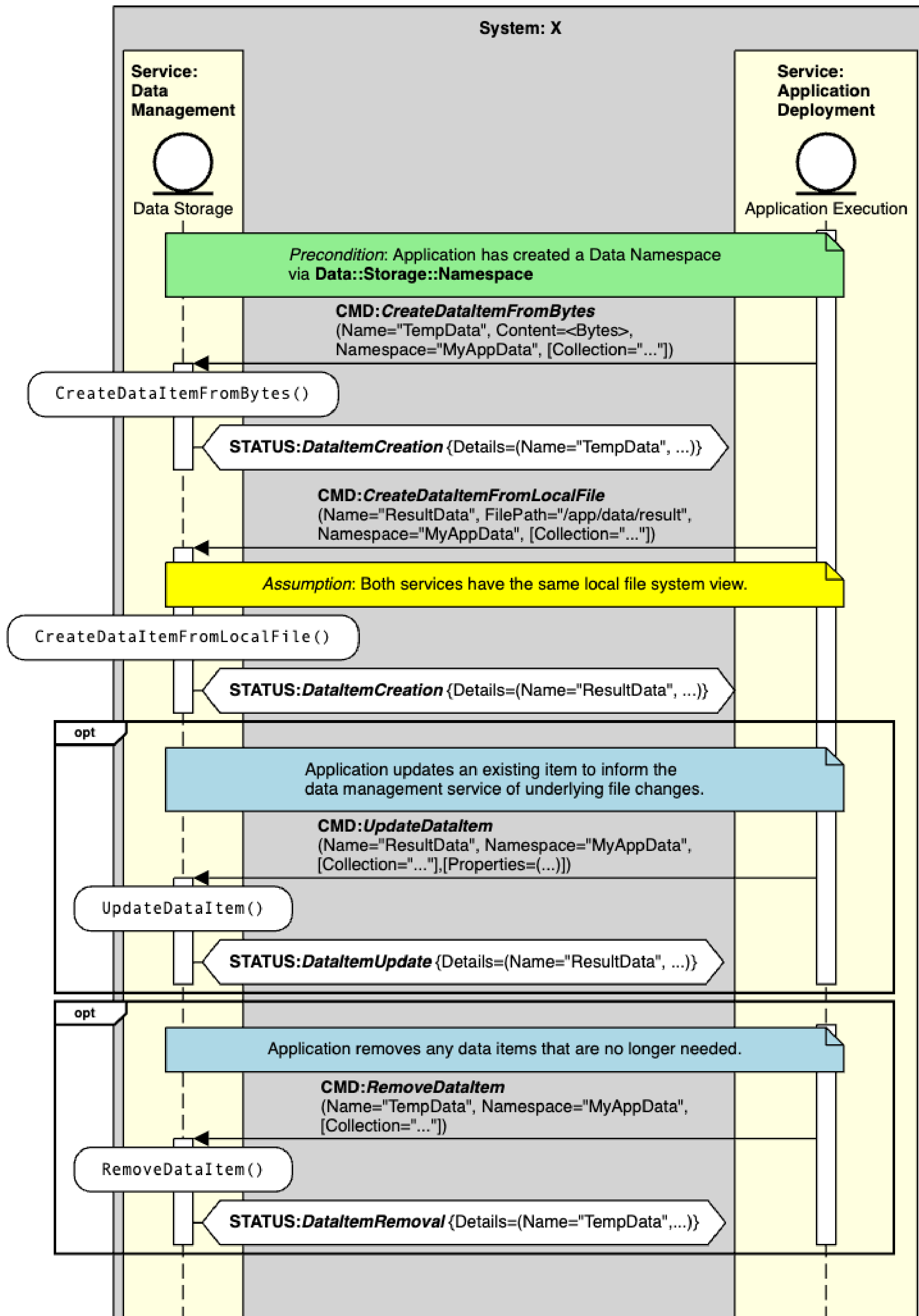

Figure A-13. Sequence for an application to manage its data items.

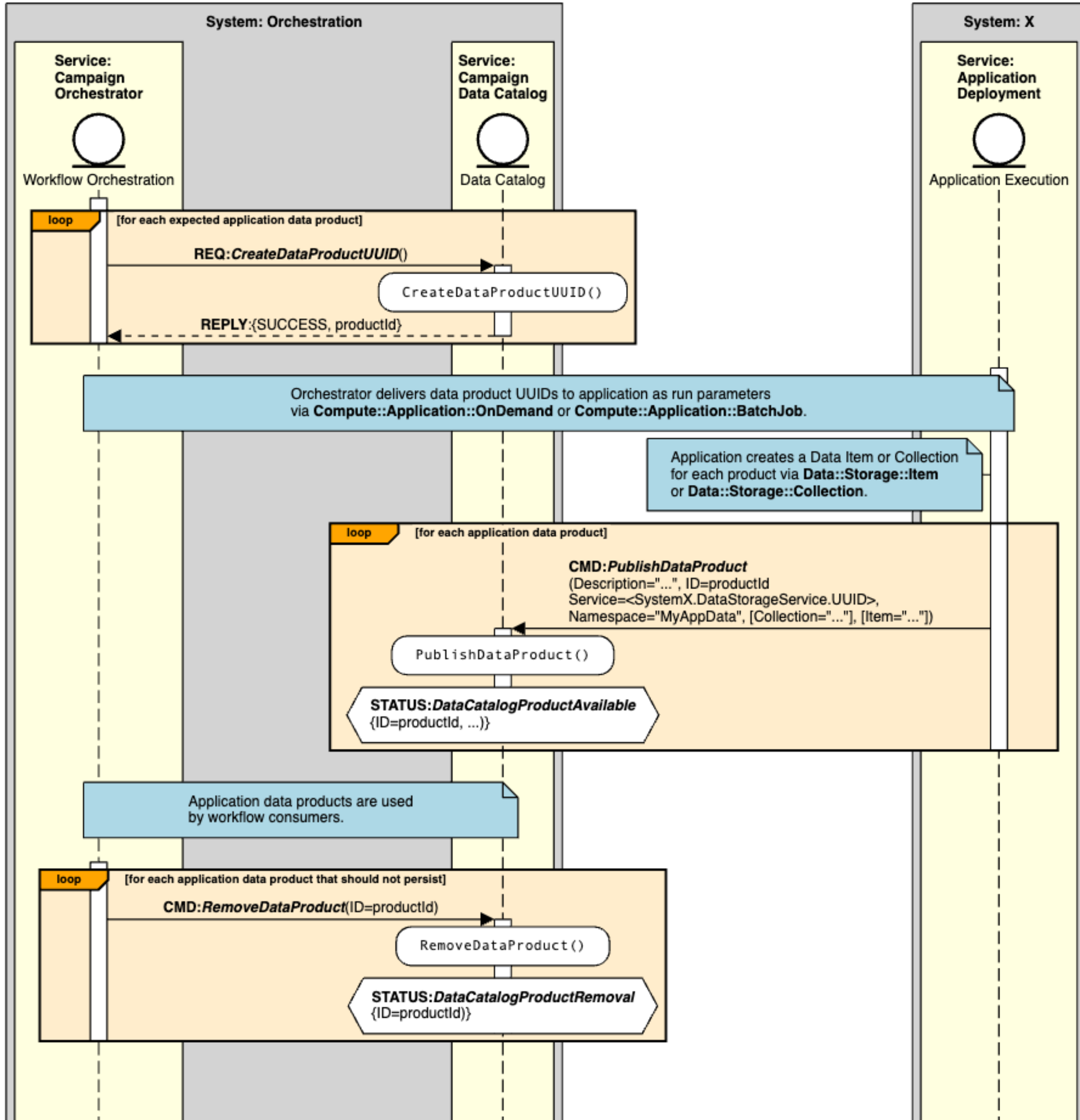

Figure A-14. Sequence for an application to publish data products.

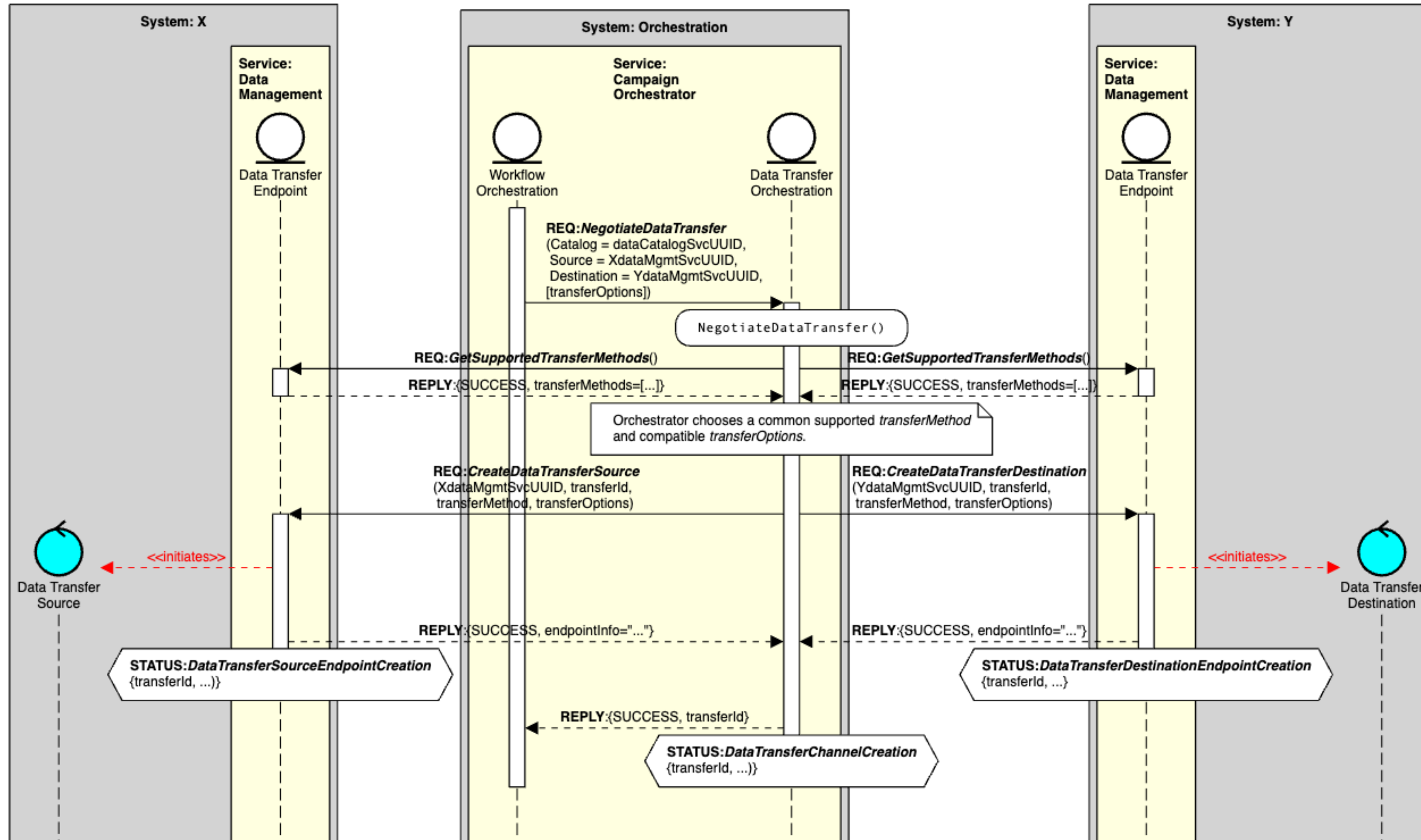

Figure A-15. Sequence for negotiating data transfers between INTERSECT systems.

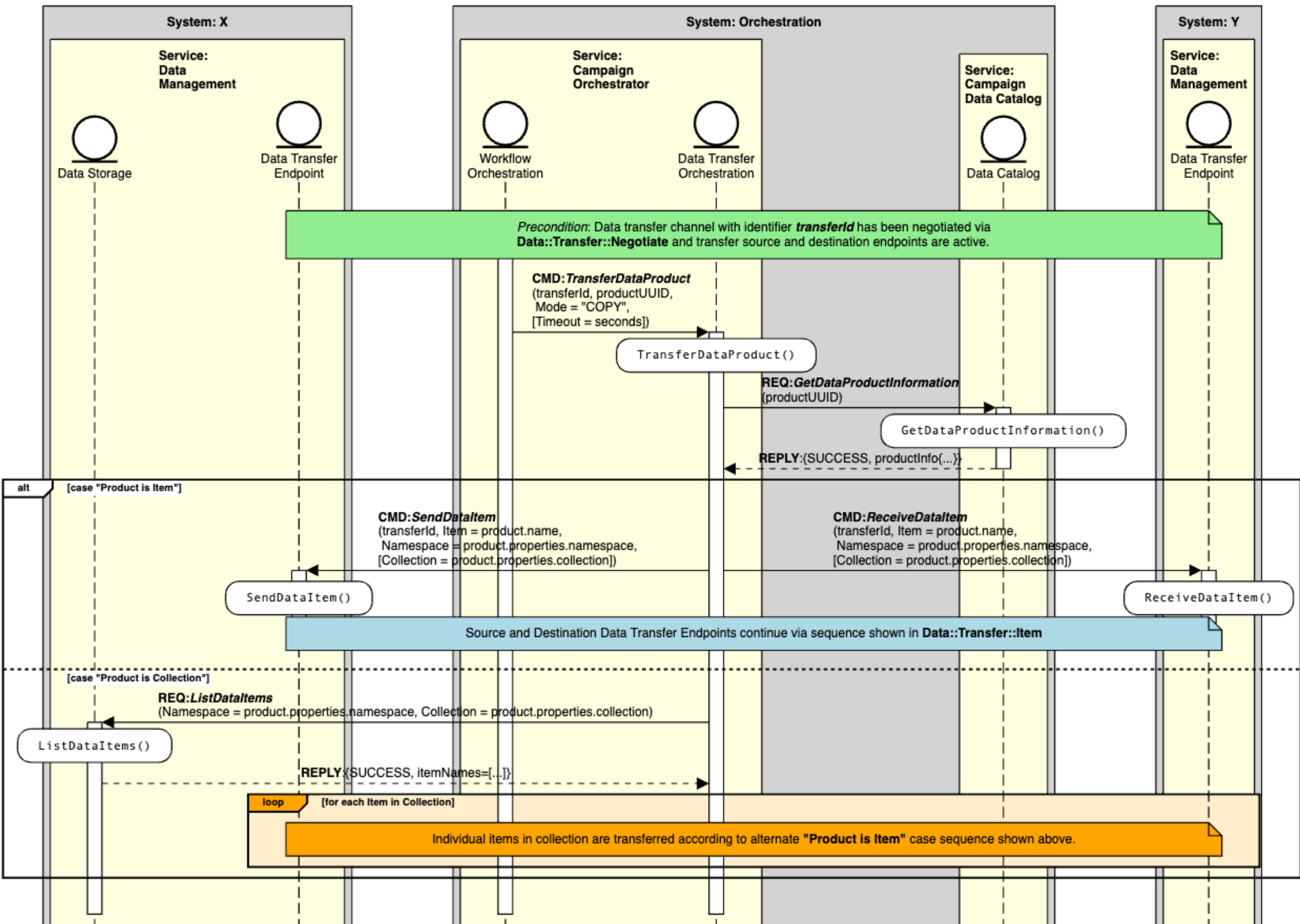

Figure A-16. Sequence for transferring a data product between INTERSECT systems.

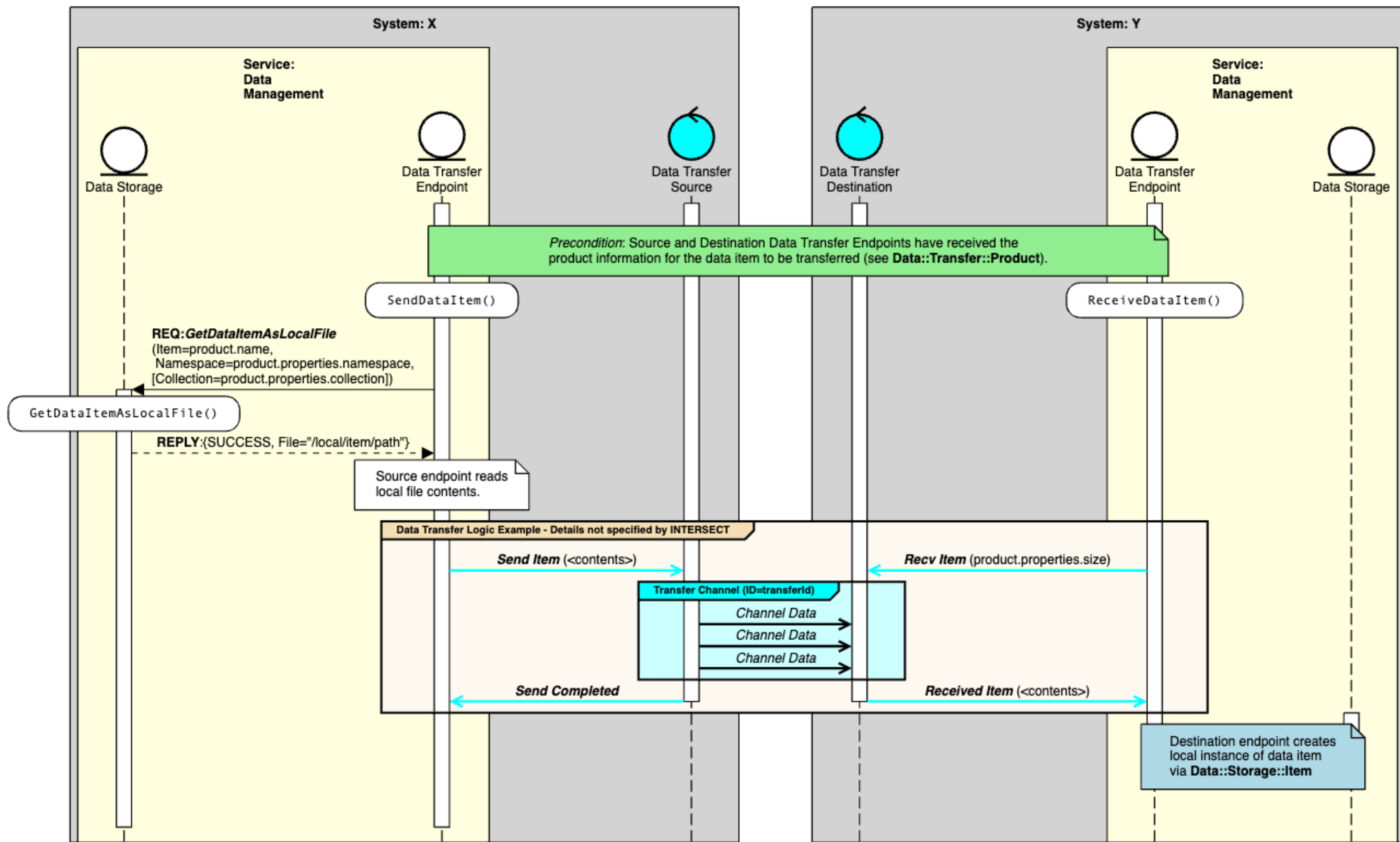

Figure A-17. Sequence for transferring a data item between INTERSECT systems.