

Joint Architecture for Unmanned Ground Systems (JAUGS)

Volume II Reference Architecture Specification

Version 1.0 Draft

developed by
Unmanned Ground Vehicles/Systems Joint Project Office
AMSAM-DSA-UG
Redstone Arsenal, Alabama 35898

Table of Contents

1. Document Organization.....	5
1.1 Overview	5
1.2 Volume I: Domain Model Description	5
1.3 Volume II: Reference Architecture Specification.....	5
1.4 Volume III: Configuration Management Plan	6
1.5 Reference Architecture Document Overview	6
2. JAUGS Requirements.....	7
3. Reference Architecture Introduction	8
3.1 Reference Architecture Requirements	8
3.1.1 Flexible Configuration	8
3.1.2 Modularity	8
3.1.3 Payload/Subsystem Reuse.....	8
3.1.4 Performance	9
3.1.5 Open Systems Standards	9
3.2 The Generic Open Architecture	9
3.3 JAUGS Physical Topology	11
3.4 JAUGS Messaging System and Components	11
3.5 Unified Modeling Language Usage	12
4. JAUGS Messaging System Overview	13
5. Message Class	15
5.1 Public Attributes:	15
5.1.1 ID : Message_ID_Type	15
5.1.2 To : Component_Address_Type	15
5.1.3 From : Component_Address_Type	15
5.1.4 Priority : Priority	15
5.1.5 Data_Size : Natural = 0	15
5.1.6 Min_Rate : Float = 0.0	15
5.1.7 Max_Rate : Float = 0.0	16
5.2 Public Operations:	16
5.2.1 Create (Header : in Header_Type, Min_Rate : in Float = 0.0, Max_Rate : in Float = 0.0, Data : in Byte_Array Message : out Message) :	16
5.2.2 ID_Is (Message : Message) : Message.ID_Type	17
5.2.3 Priority_Is (Message : Message) : Message.Priority	17
5.2.4 Destroy (Message : in Message) :	17
5.2.5 Sender_Is (Message : Message) : Component_Address_Type.....	17
5.2.6 Is_Valid (Message : Message) : Boolean.....	17
5.2.7 Destination_Is (Message : Message) : Component_Address_Type	17
5.2.8 Min_Rate_Is (Message : Message) : Float.....	17
5.2.9 Max_Rate_Is (Message : Message) : Float	17
5.3 Message Types.....	18
5.3.1 Message Router Message Type.....	18
5.3.2 Command Type Message	18
5.3.3 Query Type Messages	18
5.3.4 Inform Type Messages	18
6. Data Message Class (Derived from Message Class)	19

JAUGS Reference Architecture Specification, Version 1.0 Draft

6.1	Public Attributes:	19
6.1.1	Data : Byte_Array	19
6.2	Public Operations:	19
6.2.1	Get_Data (Message : in Message, Data : out Byte_Array)	19
7.	Message Queue Class	20
7.1	Public Attributes:	20
7.1.1	Message_Count : Natural	20
7.1.2	Type_of_Queue : Queue_Type	20
7.2	Public Operations:	20
7.2.1	Create (Type_of_Queue : in Queue_Type = FIFO, Size : in Natural, Queue : out Message Queue)	20
7.2.2	Message_Count_Is (Queue : Message Queue) : Natural	20
7.2.3	Put_Message (Queue : in Message Queue, Message : in Message, Result : out Result_Type)	20
7.2.4	Queue_Type_Is (Queue : Message Queue) : Queue_Type	20
7.2.5	Get_Message (Queue : in Message Queue, Message : out Message)	21
7.2.6	Flush (Queue : in Message Queue)	21
7.2.7	Destroy (Queue : in Message Queue)	21
8.	Message Router Class	22
8.1	Private Attributes:	22
8.1.1	Node_ID :	22
8.2	Public Operations:	22
8.2.1	Register_Component (Rank : in Rank_Type, Queue_Preference : in Message Queue.Queue_Type, ID : in Component_ID_Type, Queue : out Message Queue, Address : out Component_Address_Type Result : out Result_Type)	22
8.2.2	Register_Message (Component : in Component_ID_Type, Message : in Message) : Result_Type	22
8.2.3	Get_Registered_Message (ID : in Message.ID_Type, Message : out Message, Result : out Result_Type)	23
8.2.4	Send_Message (Message : in Message)	23
8.2.5	SC_Create (Message : in Message, Requester : in Component_Address_Type, Requested_Rate : in Float)	23
8.2.6	SC_Confirm (Provider : in Component_Address_Type, Connection : in Service Connection, Confirmed_Rate : in Float)	23
8.2.7	SC_Activate (Requester : in Component_Address_Type, Connection : in Service Connection)	24
8.2.8	SC_Suspend (Requester : in Component_Address_Type, Connection : in Service Connection)	24
8.2.9	SC_Terminate (Requester : in Component_Address_Type, Connection : in Service Connection)	24
8.2.10	SC_Request_Termination (Provider : in Component_Address_Type, Message : in Message)	24
8.2.11	SC_Kill (Provider : in Component_Address_Type, Connection : in Service Connection)	24
8.2.12	Withdraw_Message (Component : Component_ID_Type, Message : Message) : Result_Type	24
8.2.13	Withdraw_Component (Component : Component_ID_Type) : Result_Type	24
8.3	Component Registration Use Case	25
8.4	Register to Handle Message Use Case	25

JAUGS Reference Architecture Specification, Version 1.0 Draft

8.5	Locate Component to Handle Message Use Case	26
8.6	Establish Service Connection Use Case	27
8.7	Manage Service Connection Use Case	28
8.8	Withdraw Message Use Case	29
8.9	Withdraw Self Use Case.....	30
8.10	Message Router Databases	30
8.10.1	Component Database	30
8.10.2	Message Database	30
8.11	Restrictions on Message Delivery	30
8.12	Message Router Types.....	30
8.12.1	Result Type	31
8.12.2	Component_Address_Type.....	31
9.	Service Connection Class.....	32
9.1	Public Attributes:.....	32
9.1.1	Status : Status_Type.....	32
9.1.2	Rank : Rank_Type.....	32
9.1.3	Requester : Component_Address_Type.....	32
9.1.4	Provider : Component_Address_Type.....	32
9.1.5	ID : Message_ID_Type	32
9.2	Public Operations:	33
9.2.1	Create (Message : in Message, Connection : out Service Connection) :.....	33
9.2.2	Activate (Connection : in Service Connection) :	33
9.2.3	Suspend (Connection : in Service Connection) :	33
9.2.4	Read (Connection : in Service Connection, Data : out Byte_Array, Serial_Number : out Serial_Type, Time_Stamp : out Time_Type) :.....	33
9.2.5	Write (Connection : in Service Connection, Data : in Byte_Array) :.....	33
9.2.6	Rank_Is (Connection : Service Connection) : Component.Rank_Type	33
9.2.7	Status_Is (Connection : Service Connection) : Status_Type	33
9.2.8	Terminate (Connection : in Service Connection) :	34
9.2.9	Message_ID_Is (Connection : Service Connection) : Message_ID_Type	34
9.2.10	Provider_Address_Is (Connection : Service Connection) : Component_Address_Type	34
9.2.11	Requester_Address_Is (Connection : Service Connection) :.....	34
9.3	Inter-nodal Service Connection Operation	34
10.	Component Class	36
10.1	Public Attributes:.....	36
10.1.1	ID : Component_ID_Type	36
10.1.2	Rank : Rank_Type.....	36
10.1.3	State : State_Type	36
10.2	Operations.....	36

1. Document Organization

1.1 Overview

The Joint Architecture for Unmanned Ground Systems (JAUGS) is the architecture required for use in the research, development, and acquisition of Department of Defense (DoD) unmanned ground vehicle systems (UGVS). The JAUGS is defined in three separate volumes. Volume I is the JAUGS Domain Model (DM). Volume II is the JAUGS Reference Architecture (RA). Volume III is the JAUGS Configuration Management Plan (CMP).

1.2 Volume I: Domain Model Description

The JAUGS Domain Model is the model of the operational requirements, both known and potential, that may be requested by a Combat Developer (CBTDEV). The CBTDEV is the final user of the system to be acquired. The Material Developer (MATDEV) is the organization that specifies the system to be acquired and contracts with a prime contractor to have the system built. The MATDEV uses the DM to model the operational requirements. The tech base community uses the DM to understand future capabilities that may be desired by the CBTDEV. The DM is a tool used by the MATDEV and the tech base to understand the needs of the user.

1.3 Volume II: Reference Architecture Specification

The JAUGS Reference Architecture (RA) is the performance specification to implement the JAUGS DM. While the DM is written in the language of the CBTDEV, the RA is written in the language of the scientist and/or engineer. Each message and service (e.g., capability) defined in the RA can be directly traced to a DM requirement. The JAUGS RA defines a common set of messages and a message passing system to support the capabilities defined in the DM.

It is the intent of JAUGS that the DM be used to model all requirements, known and anticipated, of UGV systems. The RA will only implement those requirements in the DM that have gone through a technical evaluation process. The evaluation could be performed by the tech base, academia, or by industry. Therefore, all messages defined in the RA can be traced back to a DM service, but all DM services may not trace forward to an RA message.

The DM is used to model the CBTDEV's requirements to provide a framework for acquisition and R&D. The RA is used to specify the MATDEV's or the Tech Base's requirements to the system prime contractor (KTR). The RA is the standard against which compliance by the prime contractor will be assessed. Figure 1 shows the intended audiences for the DM and RA documents.

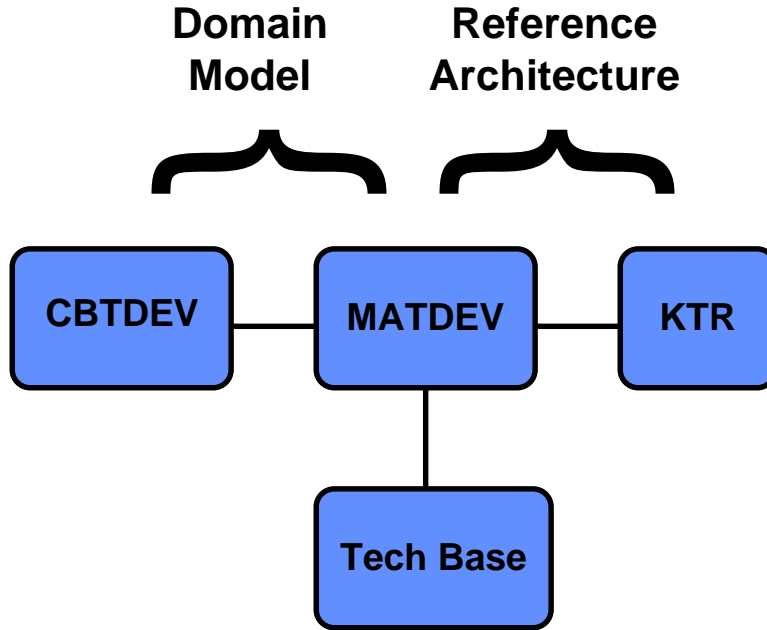


Figure 1: Scope of the JAUGS Documents

1.4 Volume III: Configuration Management Plan

The JAUGS Configuration Management Plan (CMP) is an independent document that defines the process to update the JAUGS DM and RA. The CMP establishes the charter and organization for the JAUGS Configuration Control Board (CCB). The JAUGS CCB is the governing body that is authorized to modify the DM and the RA. The JAUGS CMP also defines the process to validate compliance with the RA.

1.5 Reference Architecture Document Overview

Sections 1 through 3 provide background information for this document. Section 4 provides an overview of the JAGUS messaging system. Sections 5 through 10 describe the messaging system class definitions.

2. JAUGS Requirements

The purpose of the Joint Architecture for Unmanned Ground Systems (JAUGS) is to support the acquisition of Unmanned Ground Vehicle Systems (UGVs) by providing a mechanism for the reduction of life cycle costs and by providing a framework for technology insertion. The JAUGS documentation consists of three separate volumes: Volume I, Domain Model; Volume II, Reference Architecture; and Volume III Management Plan.

This volume, the JAUGS Reference Architecture, provides a technical specification for key components of UGVs. The Reference Architecture provides implementers with a paradigm for building systems based on standard interfaces and system services. By adhering to this specification, JAUGS compliant systems will attain a high degree of reuse and portability with other UGVs.

3. Reference Architecture Introduction

The Joint Architecture for Unmanned Ground Systems (JAUGS) Reference Architecture is an abstraction of the Unmanned Ground Vehicle System (UGVS) domain that standardizes certain features of the system without specifying their implementation - with regard to either hardware or software. The Reference Architecture is traceable to the Domain Model, providing technical direction and guidance to those who implement UGVs.

3.1 Reference Architecture Requirements

The requirements of the Reference Architecture are discussed below. Everything specified in the Reference Architecture should be traceable to one or more of these requirements. It is desirable to specify only what is required to achieve the desired result. Users of this document are encouraged to participate in the evolution of this specification. The success of JAUGS depends on the support and cooperation of the UGVS community.

3.1.1 *Flexible Configuration*

UGVS are distributed systems, utilizing processing resources that can be distributed within the vehicle, and to other elements in the system, such as an Operator Control Unit (OCU) or another UGVS. Software capabilities must be able to be distributed among the processing resources within the system. Furthermore, the software capabilities should be able to be moved from one processing resource to another with minimal impact to the system. This configuration flexibility is crucial to achieving systems that can evolve with technology and do so at a cost that is affordable.

3.1.2 *Modularity*

It is impossible to predict the various payloads that UGVS will carry in the future. The Reference Architecture must support the integration of payload modules in a generic and robust manner. In addition, UGVs must be field configurable for various missions. The requirement for reconfiguration in the field means that a UGVS will have to recognize, and utilize many different payload combinations.

3.1.3 *Payload/Subsystem Reuse*

A payload or subsystem should be able to be used in more than one UGVS. Reuse of payloads and subsystems between different UGVs is a major cost cutting and reliability goal of JAUGS. If one contractor develops a superior obstacle detection system, then it should be reused on other systems within the UGVS domain. Size, weight, power, physical connections, and other factors limit the extent to which payloads and subsystems can achieve horizontal reuse across the product line, but the Reference Architecture supports this concept from an operational and communications point of view.

JAUGS Reference Architecture Specification, Version 1.0 Draft

3.1.4 Performance

JAUGS must support the hard real-time requirements of weapon systems. Future UGVs may be called upon to integrate weapons, sensors, and other devices requiring sub-millisecond data latency and high periodic process execution rates. JAUGS must include features to support high performance real-time systems.

3.1.5 Open Systems Standards

Wherever possible, JAUGS will support the use of open standards that are based on commercial practices and open technology. Proprietary and/or contractor specific solutions that could adversely affect the flexibility, modularity, and reuse of subsystems and payloads are prohibited. Implementation details within subsystems and payloads are not affected by the Reference Architecture.

3.2 The Generic Open Architecture

The JAUGS Reference Architecture is a technical specification to support acquisition of UGVs. It supplements the Joint Technical Architecture – Army (JTA-A)¹ by providing direction in areas not covered by the JTA-A. The JTA-A includes annexes to address the specialized requirements of various system domains within the Department of Defense (DoD). The Weapon Systems Annex addresses the requirements of real-time embedded systems. The architects of the Weapon Systems Annex have adopted the Society of Automotive Engineers (SAE) Generic Open Architecture (GOA). The GOA is “a framework to identify interface classes for applying open systems to the design of a specific hardware/software system.”² Using the GOA framework as a reference, the areas of applicability of the JAUGS Reference Architecture can be identified.

¹ <http://www-jta.itsi.disa.mil/>

² Overview and Rationale for GOA Framework Standard, SAE AIR5315, July 1997.

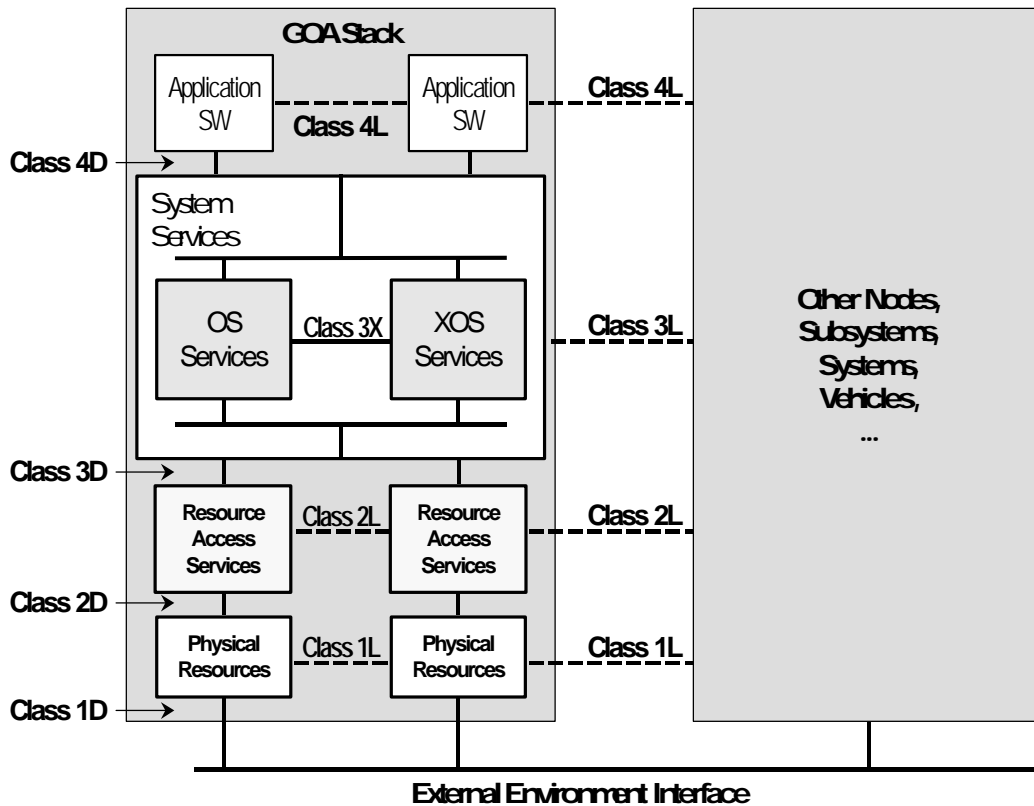


Figure 2: Generic Open Architecture Stack Diagram

Figure 2 shows the GOA Stack. The GOA Stack describes a layered system with clearly defined interfaces between layers and within layers. Direct interfaces (D suffix) are defined between layers in the architecture. Logical interfaces (L suffix) are defined across layers. Logical interfaces are peer interfaces, describing interactions between entities at the same level. For instance, at the logical level, one application software component would send a message to another application software component. Logically, the message goes from the source component to the destination component. In actuality, the message passes through the System Services layer, on to the Resource Access Services, then to a Physical Resource. At the level of the physical resource, the message is transmitted as electrical signals over a bus, or network, or some other medium.

The JAUGS Reference Architecture applies to 4L, 4D, and 3L interfaces, and defines an eXtended Operating System (XOS) service. The 4L interface specified are the message sets that are used to communicate from one application component to another. The 4D interface specified is the set of services to be provided by a standard XOS messaging system. The 3L interface is the binary representation format of messages sent between messaging system on different nodes.

As JAUGS develops and matures, the GOA will serve as the framework for classifying and incorporating open systems standards.

3.3 JAUGS Physical Topology

The physical topology of JAUGS is shown in Figure 3. A System is composed of one or more operational subsystems. Examples of an operational subsystem are a vehicle or an operator control unit. Each operational subsystem is composed of one or more nodes. A node is synonymous with a GOA Stack (see Figure 2: Generic Open Architecture Stack Diagram.) Each node has its own message system. A node is composed of one or more components. The component is the lowest level abstraction in the JAUGS Reference Architecture.

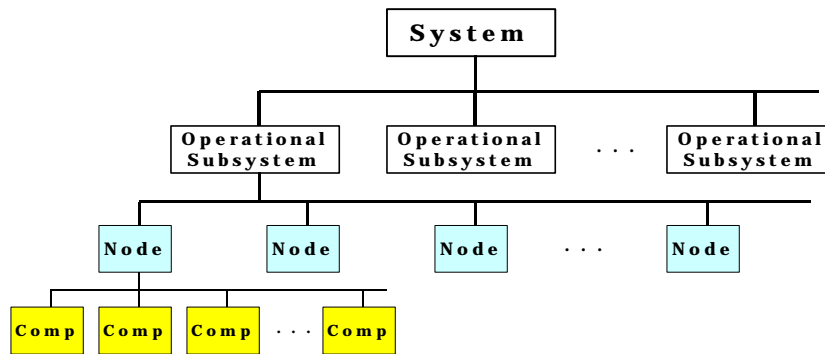


Figure 3: Physical Topology of JAUGS

3.4 JAUGS Messaging System and Components

JAUGS specifies a messaging system which provides communication services between components located within the same operational subsystem. The messaging system does not route messages between components on different operational subsystems. A specialized component, the Intra-UGV Communicator, provides communication services between operational subsystems. The Intra-UGV Communicator component is discussed in Section 10. Messages are routed between components within an operational subsystem.

The JAUGS messaging system supports two distinct methods of transmitting messages between components. The two methods of transmission are a traditional message queue (in box) and service connections. The message queue mechanism supports asynchronous communication between components. When a component sends a message to another component, the message arrives in the destination component's message queue. Service connections provide a means of synchronous, periodic data flow with minimal overhead.

JAUGS Reference Architecture Specification, Version 1.0 Draft

A service connection establishes a unidirectional link between two components for the transmission of a specific message. Service connections are expected to satisfy hard real-time requirements for high rate data signals with low latency. When a data item is required on a constant, periodic basis, then a service connection should be used.

A JAUGS compliant Unmanned Ground Vehicle System (UGVS) must dynamically adapt to various payload configurations in the field. At the Application Software layer (the top layer of the GOA stack), JAUGS partitions the system into components. Section 10 covers components in more detail, but a brief discussion is included here to support the messaging system concepts. Components are black box entities that handle messages and perform the functions of the system. A system is composed of components. JAUGS does not address how components are implemented, but does specify the interfaces between components in the system (GOA Class 4L), and the interface to the JAUGS messaging system (GOA Class 4D). A component must reside on a single node in the system. A node may contain multiple components, but a component cannot span multiple nodes. Each component should have its own thread of control in the system. Components should operate concurrently.

3.5 Unified Modeling Language Usage

The Unified Modeling Language (UML) is used in this document to describe the reference architecture. UML is a language independent, object oriented method of describing systems. The reference architecture is composed of classes. The attributes and operations of these classes are describes. The relationships between the classes are also modeled. Use Cases will be used to generically describe interactions between classes. The Use Case method is part of the UML methodology.

A recommended reference for UML is Eriksson, Hans-Erik, M. Penker, "UML Toolkit," John Wiley & Sons, Inc., 1998, ISBN: 0-471-19161-2.

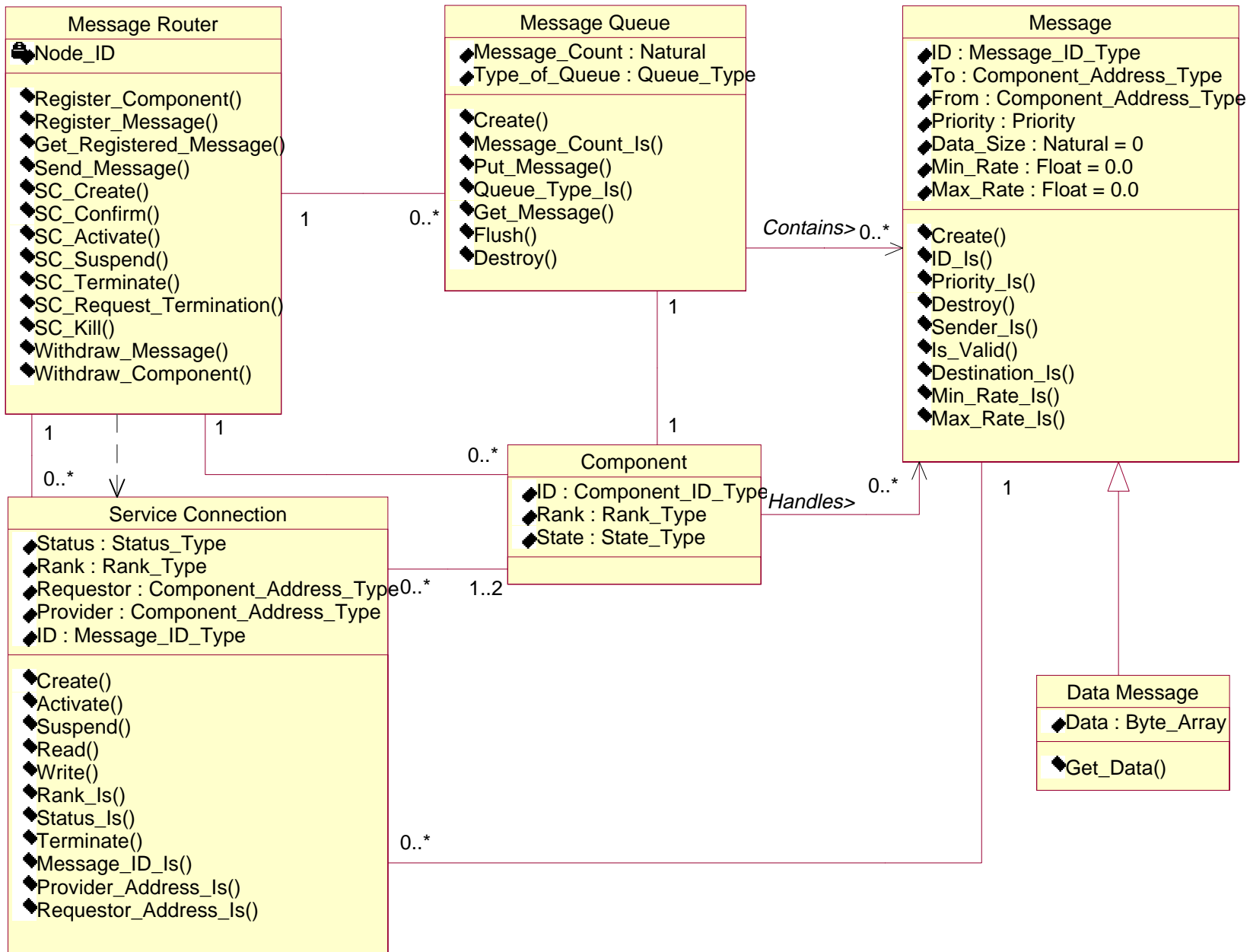
4. JAUGS Messaging System Overview

The JAUGS messaging system is composed of four main classes and one subclass. The component class interacts with these classes. Figure 4 shows the class diagram of the JAUGS messaging system. The Component class is shown in the middle of the diagram. The four main classes that make up the messaging system are: the Message class, the Message Queue class, the Message Router class, and the Service Connection class. The subclass shown is the Data Message class.

The lines between the classes indicate association, dependency, and generalization relationships. There is one dependency relationship shown between the message router class and the service connection class. This is depicted by the dashed line with the arrow pointing towards the service connection class. This relationship indicates that the message router class is dependent on the service connection class. Changes in the service connection class could affect the message router class. There is also only one generalization relationship, shown between the data message class and the message class. The generalization is shown by the solid line with the hollow triangle arrow head. This relationship indicates that data message class is a subclass of message class. Data message class inherits all of the attributes and operations of the message class. The data message class add the Data attribute and the Get_Data operation.

The remaining relationships between the classes are associations. An association simply indicates that a relationship exists between two classes. The labels at each end of the association relationship indicate cardinality. For instance, the association between message queue class and component class states that each component has one message queue, and that each message queue is associated with one component. Another example is the relationship between message router class and component class. This relationship states that a message router class is associated with zero to many components, and that a component is associated with only one message router. Some associations have an arrow at one end and no cardinality state at the non-arrow end of the association. This indicates that the association only makes sense in the direction of the arrow. For instance, the association between message queue and message class states that a message queue can contain zero to many messages. The association from message class to message queue class, however, is not meaningful.

Each class is drawn as a rectangle with three horizontal fields within it. The top field is the name of the class. The middle field contains the attributes of the class, and the bottom field lists the operations of each class. The small icon to the left of each attribute and operation represents its visibility. Except for the Node_ID attribute of the message router class, all attributes and operations in this diagram have public visibility. The small lock icon on the Node_ID attribute indicates private visibility. The operations in the diagram are abbreviated. In the sections that follow, the operations are fully described in the text.



5. Message Class

5.1 Public Attributes:

5.1.1 *ID : Message_ID_Type*

Each message in JAUGS has a unique identification. The *Message_ID_Type* in JAUGS is a two byte unsigned integer. The name space of message IDs is partitioned into JAUGS standard message IDs and experimental message IDs. The JAUGS Configuration Management Plan (Volume III) covers assignment of message IDs.

5.1.2 *To : Component_Address_Type*

The *To* attribute is the address of the component to which the message will be delivered. Component addresses are assigned by the message router class when a component registers itself.

5.1.3 *From : Component_Address_Type*

The *From* attribute is the address of the component which sent (or created) the message. Component addresses are assigned by the message router class when a component registers itself.

5.1.4 *Priority : Priority*

The *priority* attribute is used by the message queue class to order messages within a priority queue. Message queues can be First-In-First-Out (FIFO) or priority. If a message is put into a FIFO queue, the *priority* attribute is not used.

5.1.5 *Data_Size : Natural = 0*

The *Data_Size* attribute represents the number of bytes of data contained in the message in addition to the fixed header size. The base message class assumes a message with no data, only a header. The attribute is included in the base class for consistency, such that all messages, whether they contain data or not, have the same size header. The default value of *Data_Size* is 0.

5.1.6 *Min_Rate : Float = 0.0*

Min_Rate is a floating point value that represent the frequency, in Hertz, at which the message can be updated or read by the component registered to handle the message, through a service connection.

Messages that can be transmitted via a service connection must specify a minimum

JAUGS Reference Architecture Specification, Version 1.0 Draft

frequency that the service connection can sustain. If a message is not suited for transmission by service connection, then the `Min_Rate` and `Max_Rate` are both zero. If a message can be transmitted via a service connection, then the following must be true:

$$0.0 < \text{Min_Rate} \leq \text{Max_Rate}$$

If `Min_Rate` = `Max_Rate`, then the service connection can only be established at the single rate specified by both parameters. The default value of `Min_Rate` is 0.0.

5.1.7 `Max_Rate` : *Float* = 0.0

`Max_Rate` is a floating point value that represent the frequency, in Hertz, at which the message can be updated or read by the component registered to handle the message, through a service connection.

Messages that can be transmitted via a service connection must specify a maximum data rate that the service connection can sustain. If a message is not suited for transmission by service connection, then the `Min_Rate` and `Max_Rate` are both zero. If a message can be transmitted via a service connection, then the following must be true:

$$0.0 < \text{Min_Rate} \leq \text{Max_Rate}$$

If `Min_Rate` = `Max_Rate`, then the service connection can only be established at the single rate specified by both parameters. The default value of `Max_Rate` is 0.0.

5.2 Public Operations:

5.2.1 *Create* (*Header* : in *Header_Type*,
Min_Rate : in *Float* = 0.0,
Max_Rate : in *Float* = 0.0,
Data : in *Byte_Array*
Message : out *Message*) :

The create operation is stereotyped to a procedure with no return value. The first parameter, `Header`, is of type `Header_Type`. `Header_Type` is a data structure that is defined as follows:

<u>Field</u>	<u>Number of Bytes</u>
Message ID	2
To Address	2
From Address	2
Priority	1
Data Size	1

The second and third parameters are for specifying the range of the rate (frequency in Hertz) at which the message can be updated or read by the component registered to

JAUGS Reference Architecture Specification, Version 1.0 Draft

handle the message through a service connection. For messages that are not suitable for service connections, both parameters should be set to their default values of 0.0

The fourth parameter, Data, is only used for messages that have data associated with them. If the Data_Size field of the header is set to 0, then this parameter is ignored.

The last parameter, is the message object that is returned from the operation.

5.2.2 *ID_Is (Message : Message) : Message.ID_Type*

Given a message object, this operation returns the value of the ID attribute.

5.2.3 *Priority_Is (Message : Message) : Message.Priority*

Given a message object, this operation returns the value of the priority attribute.

5.2.4 *Destroy (Message : in Message) :*

The message object given to this operation will destroy itself.

5.2.5 *Sender_Is (Message : Message) : Component_Address_Type*

Given a message object, this operation returns the component address associated with the From attribute.

5.2.6 *Is_Valid (Message : Message) : Boolean*

Given a message object, this operation invokes a self-check by the message object to determine the integrity of the message object. This method of this operation could be a circular redundancy check (CRC) or some other validity check.

5.2.7 *Destination_Is (Message : Message) : Component_Address_Type*

Given a message object, this operation returns the component address associated with the To attribute.

5.2.8 *Min_Rate_Is (Message : Message) : Float*

Given a message object, this operation returns the frequency, in Hertz, of the minimum rate at which the component that handles this message can establish a service connection.

5.2.9 *Max_Rate_Is (Message : Message) : Float*

Given a message object, this operation returns the frequency, in Hertz, of the maximum rate at which the component that handles this message can establish a service connection.

5.3 Message Types

JAUGS will classify messages according to type. The message ID attribute will be used to distinguish the message type. All message types in JAUGS are currently non-blocking. The four types of message currently defined in JAUGS are discussed in the following sections

5.3.1 *Message Router Message Type*

Message Router type messages are only sent by the message router class. Components must be able to handle (understand) all messages of this type.

5.3.2 *Command Type Message*

Command type messages are used to effect system mode changes, alter the state of a component or system, or initiate an action which has safety implications. The message router class enforces a restriction on command type messages that only allows command type message to flow to an equal or lower rank than the sender.

5.3.3 *Query Type Messages*

Query type messages are used to solicit information from another component. An inform type message is usually generated in response to a query type message.

5.3.4 *Inform Type Messages*

Inform type messages allow components to transmit information to each other. Status reports, Built-In-Test (BIT) results, and aperiodic state information are some examples of inform messages.

6. Data Message Class (Derived from Message Class)

6.1 Public Attributes:

6.1.1 *Data : Byte_Array*

This attribute is the data associated with the message. The data within the message is represented as an array of bytes. The *Data_Size* attribute is the size of the array in bytes.

6.2 Public Operations:

6.2.1 *Get_Data (Message : in Message, Data : out Byte_Array) :*

Given a message object, this operation provides the message's data as an array of bytes.

7. Message Queue Class

7.1 Public Attributes:

7.1.1 *Message_Count* : *Natural*

This attribute specifies the number of message contained in the message queue.

7.1.2 *Type_of_Queue* : *Queue_Type*

This attribute specifies the type of queue that was created for the object. The current values of queue type are: FIFO (First-In-First-Out) and Priority.

7.2 Public Operations:

7.2.1 *Create* (*Type_of_Queue* : *in Queue_Type = FIFO*, *Size* : *in Natural*, *Queue* : *out Message Queue*) :

This operation create a message queue object. The type of queue desired and the size of the queue in bytes are provided as “in” parameters. The default queue type is FIFO. The size of the queue must be specified, as there is no default value. A queue object is returned as the “out” parameter.

7.2.2 *Message_Count_Is* (*Queue* : *Message Queue*) : *Natural*

Given a message queue object, this operation returns the value of the message count attribute.

7.2.3 *Put_Message* (*Queue* : *in Message Queue*, *Message* : *in Message*, *Result* : *out Result_Type*) :

Given a message queue and message object, this operation places the message object in the queue. This operation is reserved for use by the message router class. Components should not utilize this operation. If the message queue is full, this operation will fail. The “out” parameter, Result, will indicate the success or failure of the operation. If the operation fails, then the message router should generate an error message to the sender.

7.2.4 *Queue_Type_Is* (*Queue* : *Message Queue*) : *Queue_Type*

Given a message queue object, this operation returns the value of the *Type_of_Queue* attribute.

JAUGS Reference Architecture Specification, Version 1.0 Draft

7.2.5 Get_Message (Queue : in Message Queue, Message : out Message) :

This operation is used by components to retrieve messages from their message queue.

7.2.6 Flush (Queue : in Message Queue) :

This operation will cause all messages currently in the queue to be destroyed. This operation is only intended for used in exceptional circumstances.

7.2.7 Destroy (Queue : in Message Queue) :

This operation destroys a message queue object. This operation should only be performed by the message router.

8. Message Router Class

8.1 Private Attributes:

8.1.1 *Node_ID* :

The *Node_ID* is a private attribute that is used by message routers in multinode systems to perform internode communications. This version of JAUGS does not deal with multinode systems.8.2.13

8.2 Public Operations:

8.2.1 *Register_Component* (*Rank* : in *Rank_Type*,
Queue_Preference : in *Message_Queue.Queue_Type*,
ID : in *Component_ID_Type*,
Queue : out *Message_Queue*,
Address : out *Component_Address_Type*
Result : out *Result_Type*) :

This operation is used by components to register themselves with the message router. A component must register its rank, queue preference, and unique ID. Each of these three items remains static during system operation. The “out” parameters of this operation are a message queue object, and the address of the component assigned by the message router, and the result of the operation. If a component attempts to register using an ID that has already been chosen, then the result parameter will indicate that the operation failed.

8.2.2 *Register_Message* (*Component* : in *Component_ID_Type*,
Message : in *Message*) : *Result_Type*

This operation is used by a component to register itself as being able to handle a specific message. JAUGS currently only allows a message to be handled by one component within an operational subsystem. If a component attempts to register a message that has already been registered, then the return result will indicate failure.

A message object is passed into this operation, which must first be created by the component. In creating the message object, the “To” field of the message header is irrelevant to this operation. The “From” field of the header is given the address of the registering component. The *Min_Rate* and *Max_Rate* parameters are specified according to the components capabilities to provide a service connection for the message being registered.

JAUGS Reference Architecture Specification, Version 1.0 Draft

8.2.3 *Get_Registered_Message* (*ID* : in *Message.ID_Type*,
Message : out *Message*,
Result : out *Result_Type*) :

This operation is used by a component to locate another component to handle a particular message, or to provide a service connection for the specified message. Only the message ID is specified as the “in” parameter, with a message object being return, if successful. The result parameter indicates the success or failure of the operation.

If a component has registered to handle the message, and a message object is returned, then the operation of the message class are utilized to extract the pertinent data needed from the message object.

8.2.4 *Send_Message* (*Message* : in *Message*) :

Given a message object, this operation takes the message and delivers it to the component specified by the address of the “To” attribute of the message.

8.2.5 *SC_Create* (*Message* : in *Message*,
Requester : in *Component_Address_Type*,
Requested_Rate : in *Float*) :

Note: The current UML model of the JAUGS messaging system shows that the operations of the service connection class are all public. Until a proper method of modeling the desired behavior can be found, the text of this document will specify that certain operations of the service connection class are privileged, and can only be used by the message router class. The reason for this is that certain operations of the service connection class affect both the message router class and the component class. The current solution is that the message router class offers a set of operations with the SC_ prefix that are used by components to invoke the corresponding operations of the service connection class.

This operation is used by the component requesting a service connection to initiate creation of a service connection for the specified message. Section 8.6 describes the Use Case for establishing a service connection between two components. The requesting component specifies a rate for the service connection to operate at. The requested rate must be greater than or equal to the minimum rate and less than or equal to the maximum rate of the message object.

8.2.6 *SC_Confirm* (*Provider* : in *Component_Address_Type*, *Connection* : in *Service Connection*, *Confirmed_Rate* : in *Float*) :

This operation is used by the provider of the service connection to confirm that the service connection is ready to activate. When created, service connection objects are in the inactive state (add cross-reference to state diagram). The confirmed rate is the rate at

which the provider of the service connection will operate. The requirement of the confirmed rate is that it is greater than or equal to the requested rate, and that it is less than or equal to the maximum rate of the message object.

8.2.7 SC_Activate (Requester : in Component_Address_Type, Connection : in Service Connection) :

This operation changes the state of a service connection object from inactive to active.

8.2.8 SC_Suspend (Requester : in Component_Address_Type, Connection : in Service Connection) :

This operation changes the state of a service connection from active to inactive.

8.2.9 SC_Terminate (Requester : in Component_Address_Type, Connection : in Service Connection) :

This operation causes a service connection to terminate itself.

8.2.10 SC_Request_Termination (Provider : in Component_Address_Type, Message : in Message) :

If a component providing a service connection wishes for all requesters with existing connections to initiate termination, then this operation is used.

8.2.11 SC_Kill (Provider : in Component_Address_Type, Connection : in Service Connection) :

If a provider of a service connection has invoked a SC_Request_Termination operation, and, after a reasonable length of time service connections still exist, then this operation can be used to kill a service connection. This operation should only be used as a last resort.

8.2.12 Withdraw_Message (Component : Component_ID_Type, Message : Message) : Result_Type

A component can use this operation to withdraw a previously registered message. A message can only be withdrawn if no service connections exist for that message.

8.2.13 Withdraw_Component (Component : Component_ID_Type) : Result_Type

A component can use this operation to withdraw from the system. If a component has any messages registered with the message router, it cannot withdraw itself. All previously registered messages must first be withdrawn before this operation will

succeed.

8.3 Component Registration Use Case

Before a component can interact with other components in the system, it must register itself with the message router. The operation to register itself is the *Register_Component* (see section 8.2.1) operation.

There are no preconditions to this operation. Establishing a message queue is fundamental to the operation of the component. Even if, for example, the component cannot operate normally because of a device failure, the component would still need to register in order to send a message indicating its status.

When the message router receives a request from a component for registration, it creates an entry for that component in a database. All relevant state information about the component is stored. Initially, the component's ID, rank, and message queue preference are stored. As the component registers to handle messages and establishes service connections, this information will also be stored by the message router. The use cases for registering to handle messages and establishing service connections are presented in sections 8.4

If the component ID given to the message router has already been used by another component, then this operation shall fail. The message queue handle returned in this situation shall be null. The ID of a component is static during system operation.

The message router creates a message queue object on behalf of the component. The type of queue created is specified by the Queue parameter. The default queue type for a component is First-In-First-Out (FIFO). Message Queue is a class which is discussed further in Section 5.2.9 and 8.6 respectively.

The post conditions of this operation are: the component is known by the message router, its state is stored in the message router's component database, and that a message queue object is created for the component. The message queue object is used to invoke the operations of the message queue.

8.4 Register to Handle Message Use Case

Components in JAUGS work together to perform the functions of the system. Each component has a role. In order to perform its role, a component usually communicates with other components in the system. Components communicate using JAUGS compliant messages.

Different messages are used by different components during operation. JAUGS components register the messages that they handle with the message router. The operation to register to handle a message is *Register_Message* (see section 8.2.2).

The preconditions of this operation are that the component is registered with the message router, and that the message being registered is not currently registered by another component within the operational subsystem.

By registering the ability to handle a message, the component is advertising a service to the other components within the operational subsystem. The component is declaring that it is able to handle this message. In addition, if the specific message registered is suitable for transmission via a service connection, then another component can request a service connection for that message.

Each of the message router(s) within an operational subsystem maintains a message database. Each database contains a list of all registered messages within the operational subsystem. In a multiple-node system, the message routers are responsible for maintaining the consistency of the message databases.

The post condition of the successful completion of this operation is that the component is registered to handle the message. The message router's component and message databases are updated to include the message handling capability registered.

8.5 Locate Component to Handle Message Use Case

A component which requires the service of another component to handle a message must query the message router to locate a component that has registered to handle that message within the operational subsystem. Each of the message router(s) within an operational subsystem maintains a message database. Each database contains a list of all registered messages within the operational subsystem. In a multiple-node system, the message routers are responsible for maintaining the consistency of the message databases.

The operation used to determine whether a component has registered to handle a specific message is the *Get_Registered_Message* operation (see section 8.2.3).

There are no preconditions for this operation. Even if a component is not registered with the message router, it can still utilize this operation.

The post condition of this operation is that the requesting component knows whether or not another component within the operational subsystem has registered to handle the message. If the message is registered, then the requesting component is given the registered message object, from which it can determine the address of the component that handles the message, and the minimum/maximum data rate of the message, if applicable.

Resolution of circular references is not addressed in this version of JAUGS. For instance, consider three components: A, B, and C. Component A depends on a message from component B, which depends on a message from component C, which depends on a message from component A. If A waits to register its message handler until B has registered its, and B waits for C, and C waits for A, then none of the components will ever register. Methods of managing this problem have been suggested and will be

addressed in future versions of JAUGS.

8.6 Establish Service Connection Use Case

A service connection is an efficient message delivery mechanism in JAUGS that is intended for messages that are passed on a regular, periodic basis. Service connections pass a specific message independent of the components' message queues. Section 9 discusses service connections in greater detail. This use case describes how components establish a service connection.

The preconditions of this use case are that there are two or more components registered within the operational subsystem, that one of them is registered to handle the message for which the service connection is being established, and that the registered message is suitable for a service connection. The component that registers to handle the message will be called the provider. The component that asks for the service connection will be called the requester. Data flow within a service connection is unidirectional, and can flow either to or from the provider. The direction of data flow within the service connection depends on the type of the service connection. In either case, the protocol for establishing a service connection is the same.

The first operation used in establishing a service connection is the *SC_Create* operation (see section 8.2.5).

The next action that occurs in establishing the service connection is that the message router creates a service connection object. Next, the message router generates a message to the provider of the service. This message is the **SC_Create** message. The message contains the service connection object, the message ID of the service being requested, and the requested data rate.

Message:	SC.Create	
Data:	Connection :	Service_Connection
	ID :	Message_ID_Type
	Rate :	Float
	Rank :	Rank_Type

When the provider of the service receives the *SC_Create* message, it performs whatever processing is necessary to prepare for the service connection. As part of setup operations, the provider should initialize the data in the service connection buffer. When the provider completes its setup operations and is ready for the service connection to be activated, it invokes the *SC_Confirm* (see section 8.2.6) operation.

This call must contain the same service connection object provided by the *SC_Create* message. The rate parameter, however, is the actual rate that the data in the service connection will be refreshed or consumed at. The data rate requirement of the provider of the service is that the actual rate must meet or exceed the requested rate.

Upon confirmation that the service connection is ready to be activated, the message router

JAUGS Reference Architecture Specification, Version 1.0 Draft

generates a SC_Confirm message to the requester. This informs the requester component that the connection is ready for activation. This message provides the service connection object, the message ID of the service, and the actual rate at which the data will be updated or consumed.

Message:	SC_Confirm	
Data:	Connection :	Service_Connection
	ID :	Message_ID_Type
	Rate :	Float

The post conditions of this use case are: the service connection is now established and is in the inactive state, and; the component database in the message router(s) is updated for the requester and provider components to reflect the existence of the service connection object (in the inactive state). Section 9 provides a complete discussion of the service connection class.

8.7 Manage Service Connection Use Case

During operation, a service connection object is managed by the message router on behalf of the requester component. The operations that manage the service connection are part of the service connection object, but, since these operations can impact the operation of the message router, the requester is prohibited from invoking these operations directly. Therefore, while these operations are actually defined for the service connection object, they are included in the message router operations.

A service connection can be activated, suspended, or terminated. If a service connection is in the inactive state, activating it changes it to the active state. If it is in the active state, suspending it transitions it to the inactive state. Termination causes the service connection object to destroy itself. Upon creation, service connections are in the inactive state.

The operations to perform these tasks are:

The precondition of these operations is that a service connection exists.

When any of the above three functions are called by the requester of the service connection, the message router invokes the equivalent operation on the service connection object, and then sends a message to the provider of the service. The corresponding messages are: SC_Activate, SC_Suspend, and SC_Terminate (see sections 8.2.7, 8.2.8, and 8.2.9). The corresponding messages sent to the provider of the service connection are:

Message:	SC_Activate SC_Suspend SC_Terminate
Data:	Connection : Service_Connection
	ID : Message_ID_Type

The following post conditions apply if the service connection object is in the inactive

state. For the SC_Activate operation, the service connection is now in the active state. For the SC_Suspend operation, the service connection remains in the inactive state. For the SC_Terminate operation, the service connection object is destroyed.

The following post conditions apply if the service connection object is in the active state. For the SC_Activate operation, the service connection remains in the active state. For the SC_Suspend operation, the service connection is now in the inactive state. For the SC_Terminate operation, the service connection object is destroyed.

An additional post condition is: whenever one of these operations is performed, the message router component database is updated for the requester and provider components, reflecting the change in the service connection object status.

8.8 Withdraw Message Use Case

During operation, a JAUGS component can unregister its ability to handle a message. The operation to withdraw a message handler is *Withdraw_Message* (see section 8.2.12).

The preconditions of this operation are that no service connections exist for the message being withdrawn.

The post condition of this operation is that the component and message databases of the message router are updated, removing the reference to that component's ability to handle that message.

If a component wishes to withdraw a message, and service connections exist for that message, then the SC_Request_Termination operation is used. There is no precondition for this operation.

Invocation of this function causes the message router to send a message to all current requesters with service connections for that message. The receipt of this message should result in the requester initiating termination of their service connection.

Message:	SC_Request_Termination
Data:	Connection : Service_Connection
	ID : Message_ID_Type

There are no guaranteed post conditions for this operation.

It is possible that a requester of a service connection might be in an error state and is unable to terminate a service connection when requested to do so. This could lead to a lock condition, whereby the component that wishes to withdraw a message cannot do so. An operation can be invoked by a provider component to terminate a service connection. This operation is provided as a fail-safe mechanism and should only be used when the requester does not terminate a service connection within a tolerable time limit. In executing this operation, the message router invokes the terminate operation of the service connection object on behalf of the provider component. The operation is SC_Kill

(see section 8.2.11). The precondition of this operation is that the service connection object exists.

This operation results in the message router invoking the terminate operation of the service connection object. The message router then sends the SC_Terminate message to both the requester and provider components of the service connection object.

The post conditions of this operation are: the service connection object is destroyed, and the message router component database is updated to remove the service connection object from the requester and provider list of service connections.

8.9 Withdraw Self Use Case

A component can withdraw from the system. To withdraw from the system, the *Withdraw_Component* (see section 8.2.13) operation is used.

The precondition for this operation is that the component must not be registered to handle any messages.

The post condition of this operation is that the component is removed from the message router's component database.

8.10 Message Router Databases

Pending

8.10.1 Component Database

Pending

8.10.2 Message Database

Pending

8.11 Restrictions on Message Delivery

The message router shall not deliver a command type message from a lower ranking component to a higher ranking component.

8.12 Message Router Types

Pending

8.12.1 Result Type

Pending

8.12.2 Component_Address_Type

Pending

9. Service Connection Class

9.1 Public Attributes:

9.1.1 *Status : Status_Type*

The status attribute of the service connection reflects the current state of the object. It is either active or inactive.

9.1.2 *Rank : Rank_Type*

The rank of a service connection is inherited from the requesting component. The rank of a service connection is important when a control type service connection is being used. A control type service connection is a connection where the provider of the service reads the connection. The data read in the connection is typically used to control something in the system, such as an actuator. Multiple instances of control type service connections can exist simultaneously. For a control type service connection to operate properly, there must be a reliable way of determining which service connection should be read by the provider of the service.

In JAUGS, the following rules apply to control type service connections:

1. A Component who offers a control type service shall establish a separate service connection for each user.
2. The component shall utilize the data in the highest ranking active service connection.
3. If at least one active service connection exists for the control type message, then instances of that message that arrive in the components message queue shall be ignored, regardless of the rank of the sender.

9.1.3 *Requester : Component_Address_Type*

This attribute is the address of the component that requested the service connection.

9.1.4 *Provider : Component_Address_Type*

This attribute is the address of the component that provides the service connection.

9.1.5 *ID : Message_ID_Type*

This attribute is the corresponding message ID that the service connection implements.

9.2 Public Operations:

9.2.1 *Create (Message : in Message, Connection : out Service Connection) :*

The create operation brings into existence a service connection object. The message parameter, which is a message object, is used during the creation process to extract the information required to create the service connection object.

9.2.2 *Activate (Connection : in Service Connection) :*

This operation changes the state of the service connection from inactive to active. For inter-nodal service connections, the message router uses the state of the service connection to determine whether to transfer data between nodes.

9.2.3 *Suspend (Connection : in Service Connection) :*

This operation changes the state of the service connection from active to inactive.

9.2.4 *Read (Connection : in Service Connection, Data : out Byte_Array, Serial_Number : out Serial_Type, Time_Stamp : out Time_Type) :*

The read operation is used one of the components to read the data in the service connection buffer. The data is returned as an array of bytes. In addition to the data, a serial number and time stamp are also provided by the read operation.

9.2.5 *Write (Connection : in Service Connection, Data : in Byte_Array) :*

The write operation is used by a component to refresh the data in the service connection buffer. The write operation always overwrites the current data in the buffer. Whenever the write operation is invoked, the serial number and time stamp of the data are also updated.

9.2.6 *Rank_Is (Connection : Service Connection) : Component.Rank_Type*

This operation returns the rank of the requesting component of the service connection.

9.2.7 *Status_Is (Connection : Service Connection) : Status_Type*

This operation returns the state of the service connection object, whether it is active or inactive.

9.2.8 Terminate (Connection : in Service Connection) :

This operation causes the service connection to destroy itself.

9.2.9 Message_ID_Is (Connection : Service Connection) : Message_ID_Type

This operation returns the ID of the message that the service connection transmits.

*9.2.10 Provider_Address_Is (Connection : Service Connection) :
Component_Address_Type*

This operation returns the address of the provider component in the service connection.

9.2.11 Requester_Address_Is (Connection : Service Connection) :

This operation returns the address of the requester component in the service connection.

9.3 Inter-nodal Service Connection Operation

Figure 5 depicts the transfer of data from a service connection, both within a processing node, and between processing nodes. In the diagram, a service connection has a single provider (component A) , and two requesters. One of the requesters (component B) is on the same processing node as the provider, while the other (component C) is on a remote processing node.

The provider writes to the service connection. Component B reads the service connection without intervention by the message router. For component C to read the connection, however, the message routers of the two nodes must coordinate the transfer of the service connection data. Message router P must read the service connection data at the rate specified for the service connection on the remote node. It must then transmit the data to message router Q, which in turn writes the data to the service connection. Component C then reads the service connection data.

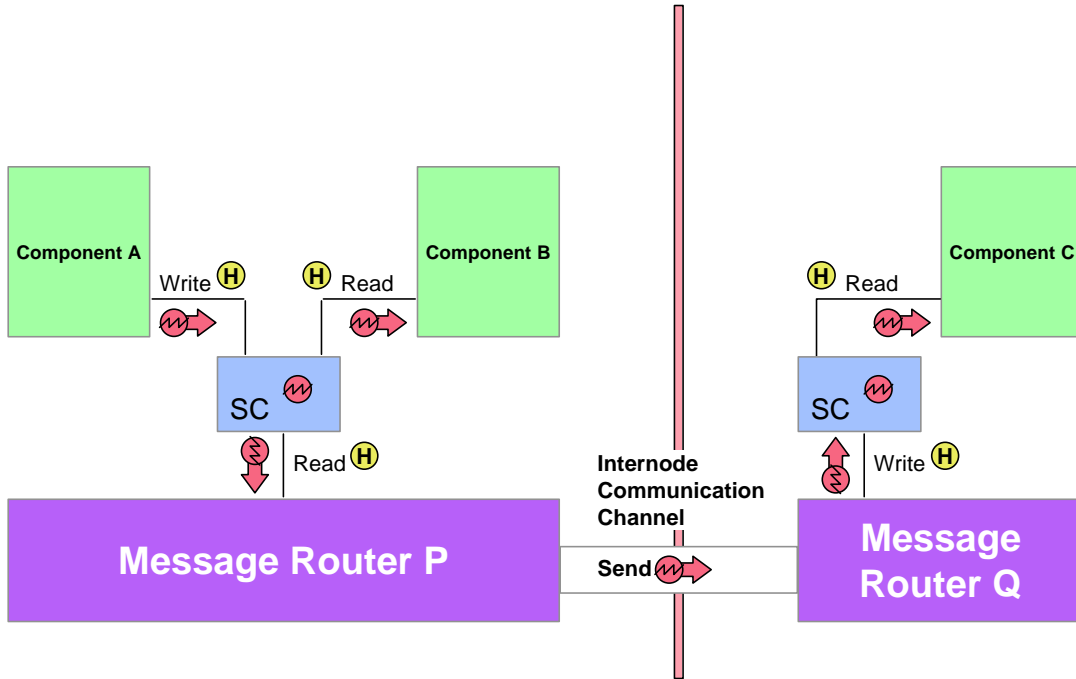


Figure 5: Intra and Inter-Node Service Connection Diagram

10. Component Class

10.1 Public Attributes:

10.1.1 *ID : Component_ID_Type*

This attribute is the unique ID of the component within the node on which it exists.

10.1.2 *Rank : Rank_Type*

This attribute is the rank of the component.

10.1.3 *State : State_Type*

This attribute is the current state of the component

10.2 Operations

The component class does not show any external operations, because all communication with components is performed through the component's message queue and service connections. Certain messages can cause the component to change state, or perform some internal operation, but these operations are not part of the class's interface specification.