**NSI Connection Service v2.0**

Status of This Document

Group Working Draft (GWD), candidate Recommendation Proposed (R-P).

Copyright Notice

Copyright © Open Grid Forum (2008-2013). Some Rights Reserved. Distribution is unlimited.

Notational Conventions

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” are to be interpreted as described in [RFC 2119].

Words defined in the glossary are capitalized (e.g. Connection). NSI protocol messages and their attributes are written in camel case and italics (e.g. *reserveConfirmed*).

Abstract

This document describes the Connection Service v2.0 which is one of a suite of services that make up the Network Service Interface.

The NSI is a web-service based API that operates between a requester software agent and a provider software agent. The full suite of NSI services allows an application or network provider to request and manage circuit service instances. Apart from the Connection Service these include the Topology Service and the Discovery Service. The complete set of NSI services is described in GFD.173 Network Services Framework.

This Connection Service document describes the protocol, state machine, architecture and associated processes and environment in which software agents interact to deliver a Connection. A Connection is a point-to-point network circuit that can transit multiple networks belonging to different providers.

Contents

[1. Introduction 3](#_Toc359491123)

[1.1 The Connection Service 3](#_Toc359491124)

[2. Network Service Framework 3](#_Toc359491125)

[2.1 NSI Services 3](#_Toc359491126)

[2.1.1 The NSI Connection Service 4](#_Toc359491127)

[2.1.2 The NSI Topology and Topology Distribution Service 4](#_Toc359491128)

[2.1.3 NSI Discovery Service 4](#_Toc359491129)

[2.2 NSA elements 4](#_Toc359491130)

[2.3 NSI architecture 5](#_Toc359491131)

[3. Describing network resources and Connections 6](#_Toc359491132)

[3.1 Topology 6](#_Toc359491133)

[3.2 Connections 7](#_Toc359491134)

[3.3 Paths 7](#_Toc359491135)

[3.3.1 Pathfinding 8](#_Toc359491136)

[3.3.2 Connection managed by an NSA chain 8](#_Toc359491137)

[3.3.3 Connection managed by an NSA tree 9](#_Toc359491138)

[4. NSI CS messages and state machines 10](#_Toc359491139)

[4.1 NSI Messages and operations 10](#_Toc359491140)

[4.2 NSI State Machines 13](#_Toc359491141)

[4.3 Data Plane Activation 16](#_Toc359491142)

[4.4 Provisioning Sequence 17](#_Toc359491143)

[5. NSI Process Coordination and Message Transport 18](#_Toc359491144)

[5.1 Message Transport 18](#_Toc359491145)

[5.2 Message and Process Coordination 19](#_Toc359491146)

[5.3 Communications 19](#_Toc359491147)

[5.4 Per Request Information Elements 19](#_Toc359491148)

[5.5 Timeouts 20](#_Toc359491149)

[5.6 ACK handling 21](#_Toc359491150)

[5.7 Message transport error handling 22](#_Toc359491151)

[5.8 Failure Recovery 22](#_Toc359491152)

[5.9 Information maintained by the Coordinator 23](#_Toc359491153)

[5.10 Per Reservation Information Elements 23](#_Toc359491154)

[5.11 Reservation Versioning Information 24](#_Toc359491155)

[5.12 Data Plane Status Information 24](#_Toc359491156)

[6. XML Schema Definitions 25](#_Toc359491157)

[6.1 Use of SOAP 25](#_Toc359491158)

[6.1.1 Asynchronous Messaging Model 26](#_Toc359491159)

[6.1.2 Synchronous Messaging Model 27](#_Toc359491160)

[6.1.3 Standard Compliance 27](#_Toc359491161)

[6.2 NSI CS Versioning 28](#_Toc359491162)

[6.3 nsiHeader element 28](#_Toc359491163)

[6.3.1 sessionSecurityAttributes type 31](#_Toc359491164)

[6.4 Common types 32](#_Toc359491165)

[6.4.1 ServiceExceptionType 32](#_Toc359491166)

[6.4.2 VariablesType 32](#_Toc359491167)

[6.4.3 TypeValuePairType 33](#_Toc359491168)

[6.4.4 TypeValuePairListType 34](#_Toc359491169)

[6.4.5 ConnectionIdType 34](#_Toc359491170)

[6.4.6 DateTimeType 34](#_Toc359491171)

[6.4.7 NsaIdType 34](#_Toc359491172)

[6.4.8 UuidType 34](#_Toc359491173)

[6.5 NSI CS operation specific type definitions. 34](#_Toc359491174)

[6.5.1 Reserve message elements 35](#_Toc359491175)

[6.5.2 reserveCommit message elements 38](#_Toc359491176)

[6.5.3 reserveAbort message elements 40](#_Toc359491177)

[6.5.4 Request: reserveAbort 40](#_Toc359491178)

[6.5.5 release message elements 43](#_Toc359491179)

[6.5.6 terminate message elements 45](#_Toc359491180)

[6.5.7 querySummary message elements 46](#_Toc359491181)

[6.5.8 querySummarySync message elements 49](#_Toc359491182)

[6.5.9 queryRecursive message elements 50](#_Toc359491183)

[6.5.10 queryNotification message elements 53](#_Toc359491184)

[6.5.11 queryNotificationSync message elements 56](#_Toc359491185)

[6.6 NSI CS specific types 57](#_Toc359491186)

[6.6.1 Complex Types 58](#_Toc359491187)

[6.6.2 Simple Types 75](#_Toc359491188)

[7. Security 77](#_Toc359491189)

[7.1 Transport Layer Security 77](#_Toc359491190)

[7.2 SAML Assertions 78](#_Toc359491191)

[8. Contributors 78](#_Toc359491192)

[9. Glossary 78](#_Toc359491193)

[10. Intellectual Property Statement 80](#_Toc359491194)

[11. Disclaimer 80](#_Toc359491195)

[12. Full Copyright Notice 80](#_Toc359491196)

[13. Appendix A: State Machine Transition Tables 80](#_Toc359491197)

[14. Appendix B: Error Messages and Best Practices 82](#_Toc359491198)

[14.1 Error Messages 82](#_Toc359491199)

[14.2 NTP servers 83](#_Toc359491200)

[15. Appendix B: Firewall Handling 83](#_Toc359491201)

[16. Appendix B: Formal statement of coordinator 85](#_Toc359491202)

[16.1 Aggregator NSA 85](#_Toc359491203)

[16.1.1 Processing of NSI Requests 85](#_Toc359491204)

[16.1.2 Requests from State Machines 86](#_Toc359491205)

[16.2 Ultimate PA 87](#_Toc359491206)

[16.2.1 Processing of NSI Requests 87](#_Toc359491207)

[16.2.2 Requests from State Machines 88](#_Toc359491208)

[17. References 89](#_Toc359491209)

# Introduction

## The Connection Service

This Open Grid Forum document defines the NSI Connection Service (CS) protocol that enables the reservation, creation, management and removal of Connections. To deliver Connection services NSI supports authentication and authorization mechanisms.

NSI is designed to support the creation of circuits (called Connections in NSI) that transit several networks managed by different providers. Traditional models of circuit services and control planes adopt a single very tightly defined data plane technology, and then hard code these service attributes into the control plane protocols. Multi-domain services need to employ heterogeneous data plane technologies. The NSI supports an abstracted notion of a Connection, and the NSI messages include a flexible schema for specifying service specific constraints. These service constraints will be evaluated against the technology available to local network service providers traversed by the service. It is up to the pathfinder of the NSI to create a path that meets these constraints. In this way the NSI allows a single Service Plane protocol suite to deliver Connections that traverse heterogeneous transport technologies.

# Network Service Framework

The CS protocol is one of the protocols that make Network Service Interface (NSI); the CS works together with these NSI services to deliver an integrated NSI service framework.

The NSI framework and architecture are normatively defined in OGF GFD.173: Network Service Framework v1.0 [1]. The NSI framework and architecture are summarized here (Section 2) for information purposes only.

## NSI Services

Network resources and capabilities are presented to the consumer through a set of Network Services, the NSF presents a unified model for interacting with these services. The NSI operates between a software agent requesting a network service and the software agent providing that Network Service. Network Services include the ability to create Connections (the Connection Service), to share topologies (the Topology Service) and to perform other services needed by a federation of software agents (the Discovery Service).

### The NSI Connection Service

The NSF includes the NSI Connection Service (CS) as one of the key NSI services. The Connection Service allows point-to-point Connections to be managed. This service is the subject of this Grid Forum Document.

### The NSI Topology and Topology Distribution Service

The NSI topology schema used by the NSI CS (and in future other NSI services) is described in: GWD-R-P: Network Service Interface Topology Representation [3]. The document describes a normative extension to the Network Markup Language (NML) base schema version 1 which allows the description of service plane objects required for the NSI CS.

The NSI Topology Distribution Service is also used by the NSI CS. This service is defined in GWD-I ‘Network Service Interface, Topology Service Distribution Mechanisms’ [2]. The document describes a normative schema which allows the description of service plane objects required for the NSI CS. Additionally it describes a set of distribution mechanisms for the network topology descriptions.

### NSI Discovery Service

The NSI discovery service is a web service that allows an NSI RA to discover information about the services available in a PA and the versions of these services. The NSI Discovery Service is defined in: GFD.xxx Network Service Interface Discovery Service [ref].

## NSA elements

The NSF describes a set of architectural elements that make up the NSI architecture; this provides a framework that applies to all of the NSI services.

The NSI provides and environment within which network resources are treated as explicitly manageable objects. Within the Network Service Framework, these network resources can be selected, allocated, queried, and managed by using NSI messages between NSAs.

The Network Service Agents (NSAs) are the agents that manage service requests. Figure 1 shows the functions that make up the NSAs.

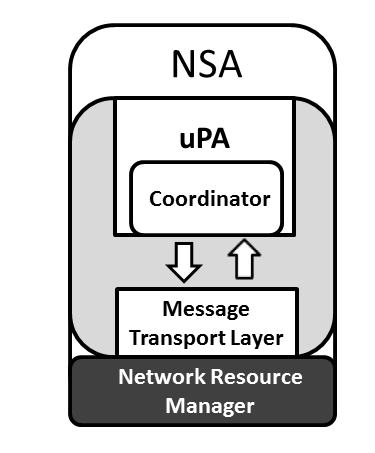


Figure : Network Service Agent

The NSA includes a logical function called the Message Transport Layer (MTL), the purpose of the MTL is to provide message delivery mechanism to the NSI layer. The message delivery layer allows has been formalized to allow the transport delivery layer to be readily changed if necessary.

The NSA includes a logical function known as a Coordinator. The Coordinator function has the role of providing intelligent message and process coordination which includes the tracking and aggregation of messages, replies and notifications and the servicing of query requests.

The NSA includes a logical function called the Network Resource Manager (NRM). The role of the NRM is to manage the resources in the Data Plane. Typically this might be an equipment vendor’s network management system.

## NSI architecture

An NSA can act either in the role of a Requester Agent (RA) or a Provider Agent (PA). NSI is an interface between these software agents and NSI Protocol messages are exchanged over the NSI interface. The Network Service Plane is defined as a notional plane in which the agents and NSI protocol reside.

The roles of the NSI agents and the NSI interface between of these agents make up the NSI architecture. NSI agents and the NSI interface are defined in by NSI framework to exist in a notional Service Plane. Each Network in the Data Plane has an associated NSA in the Service Plane.

An NSA can take on the following roles:

* uRA: The ultimate Requester Agent is the originator of a service request.
* AG: The Aggregator has more than one child NSA, and has the responsibility of aggregating the responses from each child NSA.
* uPA: The ultimate Provider Agent services requests by coordinating with the local Network Resource Manager (NRM) to manage network resources.

Within the Service Plane all NSAs have one or more peering relationships; a RA sends a service request to a PA. The NSI protocol is made up of messages that are exchanged over this peering interface.

Figure 2 below shows an example of the hierarchical relationship between NSI agents.

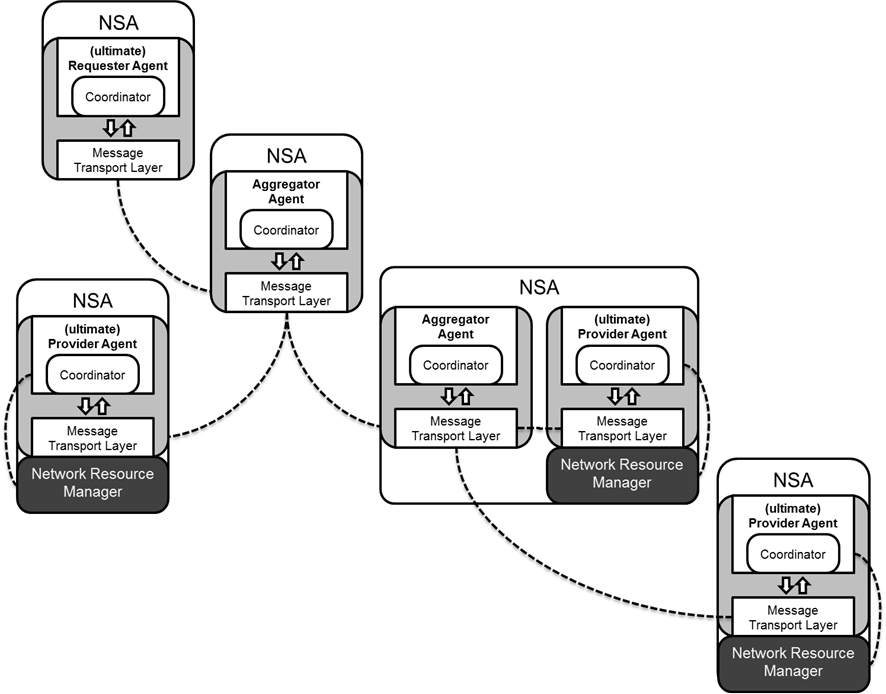


Figure : Flexible hierarchical NSA relationship

Central to the NSI architecture is the decoupling of the Service Plane from the Data Plane. NSI messages do not need to transit the NSA/Networks pairings in the same sequence that the Connection itself transits the data plane.

# Describing network resources and Connections

Section 3 of this document describes how NSI represents network resources and how the Path of a Connection can be built using this representation. The NSI topology, Connections and Paths are normatively defined in OGF GFD.173: Network Service Framework v1.0 [1]. The NSI framework and architecture are summarized here (Section 3) for information purposes only.

The CS protocol supports the creation of Connections. In the Data Plane an NSI Connection is a physical circuit through which data is delivered from an ingress point to an egress point. NSI CS v 2.0 only supports point-to-point connectivity; Connections can be unidirectional or bidirectional.

## Topology

In the NSI Topology the data plane is modeled as interconnected Networks, were the Networks are groupings of STPs. Figure 3 shows how NSI Networks interconnect at a shared point known as an SDP. An SDP is a grouping of two STPs belonging to adjacent Networks. An overview of the NSI topology concepts are introduced in GFD.173 [1] and the detailed NSI topology representation can be found in GWD-R-P: [3].

## Connections

Inter-Network Connections extend across multiple networks; they are constructed by concatenating Connections segments built across the individual networks. This is done by choosing appropriate STPs such that the egress STP of one segment corresponds directly with the ingress STP of the successive connection segment. Figure 1 shows two Networks (Y and Z) and a Connection made up of two segments (STP a - STP b) and (STP c - STP d). The inter-Network representation of the Connection maps to a physical instance in the Data Plane.

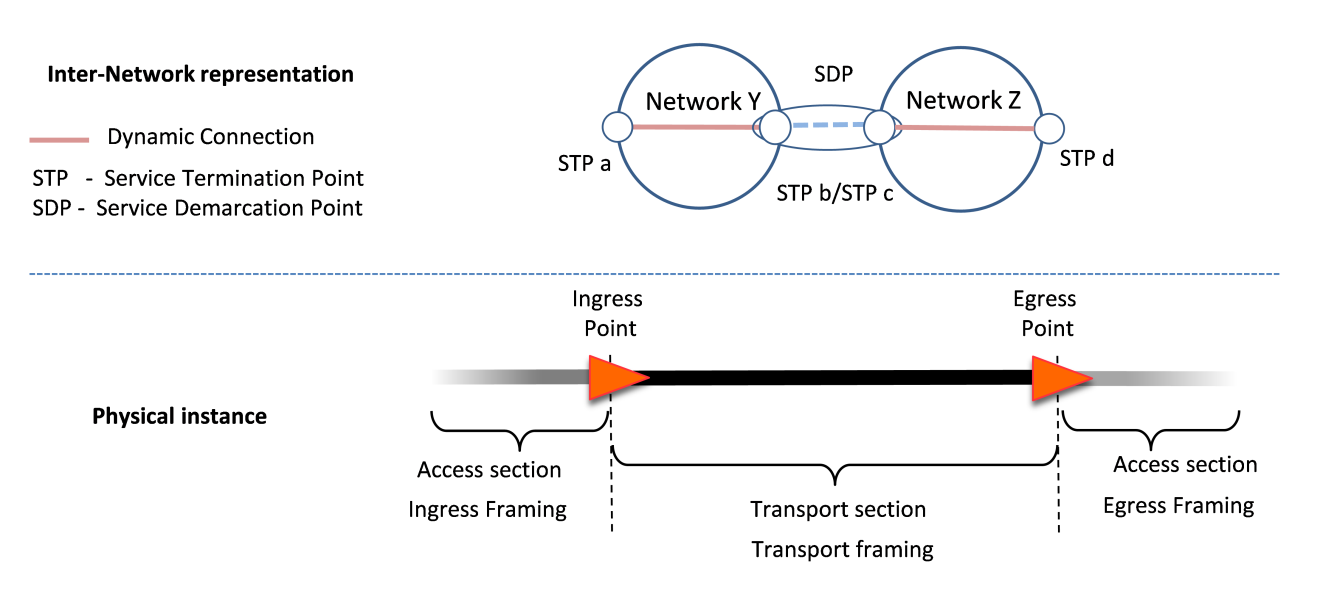


Figure : Inter-Network representation of a Connection

## Paths

A Connection request can optionally include a Path attribute. A Path is an ordered list of STPs that describe the route that should be taken by the Connection. The STPs listed in a Path will be used as constraints by the inter-Network pathfinder. The Connection will include all of the STPs in the Path in the sequence that they are listed. However a Path is not ‘strict’ in the sense that Connection is allowed to transit intermediate STPs between the STPs listed in the Path.

Figure 4 Shows an example of a Connection. This Connection could legitimately include any of the following Path: (STP a, STP b, STP d, STP f), or (STP c, STP e, STP g, STP h). Note that as the ingress and egress STPs as they are defined in the other fields of Connection Request they are not required in the Path, but may be included if desired.

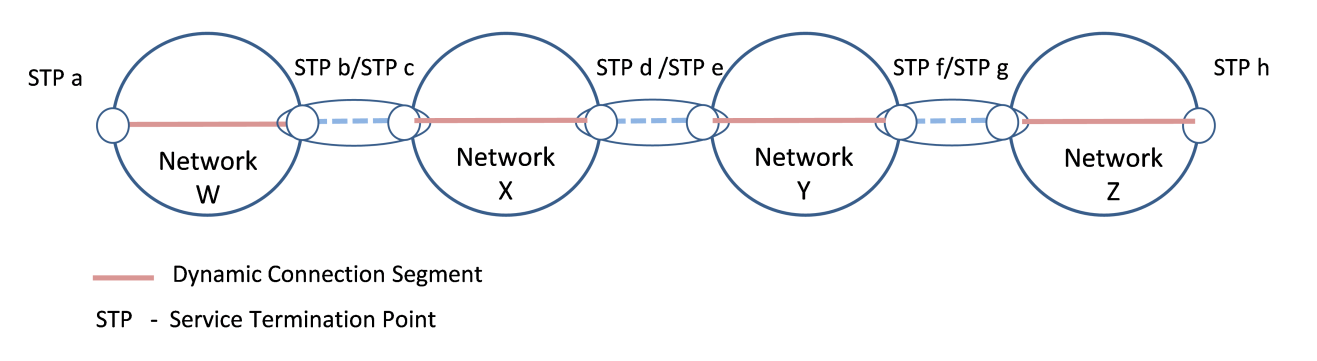


Figure : Path example

The CS does not require that NSI messages are forwarded through the same sequence of NSAs/Networks that the Connection transits, as a consequence both tree and chain type architectures are supported.

### Pathfinding

There are two levels of pathfinding related to the NSI architecture: the inter-Network pathfinding which is concerned with choosing a path across the global topology of Networks, and the intra-Network pathfinding concerned with the transport resources within the Network. These are depicted in Figure 5 below. NSI is concerned only with inter-Network pathfinding. The intra-Network pathfinding is delegated to the NRM.

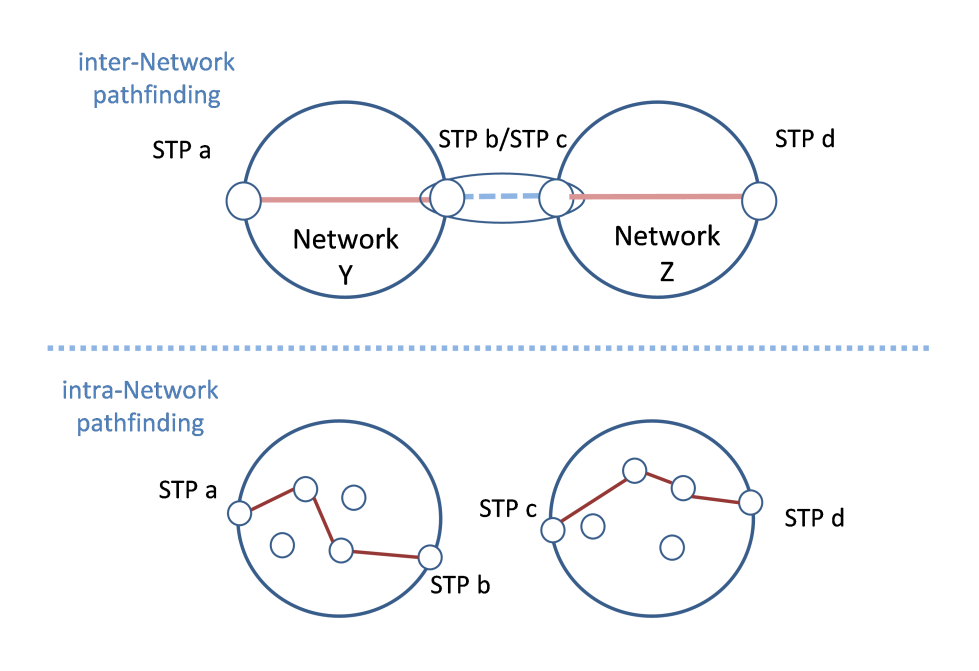


Figure : Pathfinding types

In CS v2.0 the requester will initially assume that any two STP within a particular Network can potentially be connected. If for any reason (such as fully utilized links) the requested Network internal Connection segment is unavailable, the NRM can report that intra-Network pathfinding has failed to find the needed resources.

### Connection managed by an NSA chain

Figure 6 shows an example of a Connection managed by a NSA chain. Each NSA is associated with one Network as an NSA/Network pairing. In this case the NSI message is forwarded between NSAs in the same sequence as the Connection transits the Networks.

path_chain

Figure : Example of Connection managed by a NSA chain

This example shows a Topology consisting of 3 Networks, one per NSA. This topology has the following STPs: Network X (STP a, STP b), Network Y (STP c, STP d, STP e), Network Z (STP f, STP g)

Here the NSAs are connected as a chain: Application NSA to NSA-X, NSA-X to NSA-Y and NSA-Y to NSA-Z

Assuming a Connection request comes from the Application-NSA to NSA-X to reserve a Connection STP a to STP g, then NSA-X will look in the topology and determine that to make this Connection, NSA-X will reserve a local connection from STP a to STP b, and then NSA-X forwards a request for the remainder of the connection to NSA-Y: STP c to STP g.

NSA-Y gets this request and reserves a Connection between STP c and STP e and requests a Connection from NSA-Z from STP f to STP g.

### Connection managed by an NSA tree

Figure 7 shows an example of a Connection managed by a NSA tree. In this case the NSI message is forwarded between NSAs in a different sequence compared to the sequence that the Connection transits the Networks.

path_tree

Figure : Example of a Connection managed by a NSA tree

The topology remains the same as for the previous example: Network X (STP a, STP b), Network Y (STP c, STP d, STP e), Network Z (STP f, STP g)

Here the NSAs are connected as a tree: Application NSA to NSA-X, NSA-X to NSA-Y and NSA-X to NSA-Z

Assuming a Connection request comes from the Application-NSA to NSA-X to reserve a Connection STP a to STP g, then NSA-X will look in the topology and determine that to make this Connection, NSA-X will reserve a local connection from STP a to STP b, and then NSA-X forwards a request for the remainder of the connection to NSA-Y: STP c to STP e and to NSA-Z from STP f to STP g.

# NSI CS messages and state machines

Section 4 of this document describes the messages and state machines that make up the NSI Connection Service and forms a normative part of the NSI Connection Service protocol definition.

## NSI Messages and operations

NSI messages are classified into two types, messages that are passed from an RA to a PA and messages that are passed from a PA to an RA. In addition messages can be either synchronous or asynchronous. These messages classifications are summarized Table 1Table 1 – Message types. A list of CS messages is also provided in Table 2 and Table 3.

Asynchronous requests are sent from an RA to a PA; the PA is expected to send an asynchronous reply to each request. Synchronous requests are sent from an RA to a PA; the response attributes are included in the SOAP response.

|  |  |  |
| --- | --- | --- |
| **Message type** | **Direction** | **Description** |
| Asynchronous Request | RA to PA | An asynchronous response is expected. |
| Synchronous Request | RA to PA | The response attributes are expected in the Synchronous SOAP response. |
| Asynchronous Response | PA to RA | This message is sent asynchronously in response to an asynchronous request |
| Asynchronous Notification | PA to RA | This message is sent spontaneously from a PA. |

Table – Message types

Each message invokes a corresponding operation in the recipient. The Type field in each message denotes the message type. See Section 4.2 for a description of the state machines

* If the message is of type RSM then the message is to be processed using the Reservation State Machine (RSM).
* If the message is of type PSM the message is to be processed using the Provision State Machine (PSM).
* If the message is of type LSM the message is to be processed using the Lifecycle State Machine (LSM).
* If the message is of type Query this designates a Query request and requires an associated reply message. And query Notification
* If the message is of type Notification this designates asynchronous notification messages sent by a PA to an RA.

Table 2 below summarizes the entire set of RA to PA messages. See Section 6 for details for a detailed description of these messages and their attributes.

|  |  |  |  |
| --- | --- | --- | --- |
| **NSI CS Message** (abbreviation) | **SM** | **Synch. /Asynch.** | **Short Description** |
| ***reserve***  (rsv.rq) | RSM | Asynch | The *reserve* message allows an RA to send a request to reserve network resources to build a Connection between two STP's. |
| ***reserveCommit***  ***(***rsvcommit.rq) | RSM | Asynch | The NSI CS *reserveCommit* message allows an RA to request the PA commit a previously allocated Connection reservation or modify an existing Connection reservation. |
| ***reserveAbort***  (rsvabort.rq) | RSM | Asynch | The NSI CS *reserveAbort* message allows an RA to request the PA to abort a previously requested Connection or modify a Connection that was made using the *reserve* message. |
| ***provision***  (prov.rq) | PSM | Asynch | The NSI CS *provision* message allows RA to request the PA to transition a previously requested Connection into the Provisioned state. A Connection in Provisioned state will activate associated data plane resources during the scheduled reservation time. |
| ***release***  (release.rq) | PSM | Asynch | The *release* message allows an RA to request the PA to transition a previously requested Connection into Released state. A Connection in a Released state will deactivate the associated resources. |
| ***terminate***  (term.rq) | LSM | Asynch | The *terminate* message allows an RA to request the PA to transition a previously requested Connection into Terminated state. A Connection in Terminated state will release associated resources and allow the PA to clean up the RSM, PSM and all related data structures. |
| ***querySummary***  () | Query | Asynch | The *querySummary* message provides a mechanism for an RA to query the PA for a set of Connection instances between the RA-PA pair. This message can also be used as a Connection status polling mechanism. |
| ***queryRecursive***  () | Query | Asynch | The *queryRecursive* message provides a mechanism for a RA to query the PA for a set of Connection Service reservation instances. The query returns a detailed list of reservation information collected by recursively traversing the reservation tree. |
| ***querySummarySync***  () | Query | Synch | The *querySummarySync* message is sent from a RA to a PA. Unlike the *querySummary* operation, the *querySummarySync* is synchronous and will block further message processing until the results of the query operation have been collected. |
| ***queryNotification***  () | Query | Asynch | The *queryNotification* message is sent from a RA to a PA to retrieve a list of notification messages against an existing reservation residing on the PA. The returned results will be a list of notifications for the specified *connectionId*. |
| ***queryNotificationSync***  () | Query | Synch | The *queryNotificationSync* message is sent from a RA to a PA to retrieve a list of notification messages associated with a *connectionId* on the PA. Unlike the *queryNotification* operation, the *queryNotificationSync* is synchronous and will block until the results of the query operation have been collected. |

Table 2 – RA to PA Connection Service messages

Table 3 below summarizes the entire set of PA to RA messages. See Section 6 for details for a detailed description of these messages and their attributes.

|  |  |  |  |
| --- | --- | --- | --- |
| **NSI CS Message** (abbreviation) | **SM** | **Synch. /Asynch.** | **Short Description** |
| ***reserveConfirmed***  (rsv.cf) | RSM | Asynch | The *reserveConfirmed* message is sent to the RA that issued the original *reserve* request to indicate a successful operation in response to the *reserve* request. |
| ***reserveFailed***  (rsv.fl) | RSM | Asynch | The *reserveFailed* message is sent to the RA that issued the original *reserve* request message if the requested reservation criteria could not be met. |
| ***reserveCommitConfirmed***  (rsvcommit.cf) | RSM | Asynch | The *reserveCommitConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to the *reserveCommit* request of a Connection previously in the Reserve Held state. |
| ***reserveCommitFailed***  (rsvcommit.fl) | RSM | Asynch | The *reserveCommitFailed* message is sent to the RA that issued the original request as an indication of a failure of the *reserveCommit* request. |
| ***reserveAbortConfirmed***  (rsvabort.cf) | PSM | Asynch | The *reserveAbortConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *reserveAbort* request. |
| ***provisionConfirmed***  (prov.cf) | PSM | Asynch | The *provisionConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *provision* request. |
| ***releaseConfirmed***  (release.cf) | PSM | Asynch | The *releaseConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *release* request. |
| ***terminateConfirmed***  (term.cf) | LSM | Asynch | The *terminateConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *terminate* request. |
| ***querySummaryConfirmed***  () | query | Asynch | The *querySummaryConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *querySummary* request. |
| ***querySummaryFailed***  () | query | Asynch | The *querySummaryFailed* message is sent to the RA that issued the original request as an indication of a failed operation in response to a *querySummary* request. A *querySummary* operation that results in no matching Connection does not result in a *querySummaryFailed* message, but instead a *querySummaryConfirmed* with an empty list of Connection. |
| ***queryRecursiveConfirmed***  () | query | Asynch | The *queryRecursiveConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *queryRecursive* request. |
| ***querySummarySync Confirmed***  () | query | Synch | The *querySummarySyncConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *querySummarySync* request. |
| ***errorEvent***  () | notification | Asynch | The *errorEvent* notification is raised when a fault is detected. The message includes attributes that describe an exception and include the identifier of the NSA generating the exception and the error identifier for each known fault type. The service exception supports a list of service exceptions capturing failures within the request tree. |
| ***reserveTimeout***  () | notification | Asynch | The *reserveTimeout* notification is sent to the RA that issued the original *commit* request to notify the RA that a request timeout has occurred at a PA. |
| ***dataPlaneStatusChange***  (*dataPlaneStatus*Change.nt) | notification | Asynch | The *dataPlaneStatusChange*notification is sent to the RA that issued the original *reserve* request when the data plane status has changed. Possible data plane status changes are: activation, deactivation and activation version change. |
| ***messageDeliveryTimeout***  () | notification | Asynch | The *messageDeliveryTimeout* notification is sent to the RA that issued the original *reserve* message when the delivery of a request message has timed out. |
| ***queryNotificationConfirmed***  () | query | Asynch | The *queryNotificationConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *queryNotification*request. |
| ***queryNotificationFailed***  () | query | Asynch | This *queryNotificationFailed* message is sent from the PA to RA as an indication of a *queryNotification* operation failure. |
| ***queryNotificationSyncConfirmed***  () | query | Synch | The *queryNotificationSyncConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *queryNotificationSync* request. |
| ***queryNotificationSyncFailed***  () | query | Synch | This *queryNotificationSyncFailed* message is sent from the PA to RA as an indication of a *queryNotificationSync* operation failure. |

Table 3 – PA to RA Connection Service messages

## NSI State Machines

The behavior of the NSI CS protocol is modeled in two ways: with state machines and with behavioral description of the coordinator function. In total there are three state machines, the Reservation State Machine (RSM), the Provision State Machine (PSM) and the Lifecycle State Machine (LSM). The state machines explicitly regulate sequence by which messages are processed. The CS messages are each assigned to one of the three state machines: RSM, PSM and LSM.

The sequence of operations related to RSM messages MUST conform to the Reservation State Machine shown in Figure 8. The abbreviated forms of the messages and explanations of each message are provided in Table 2 and Table 3.

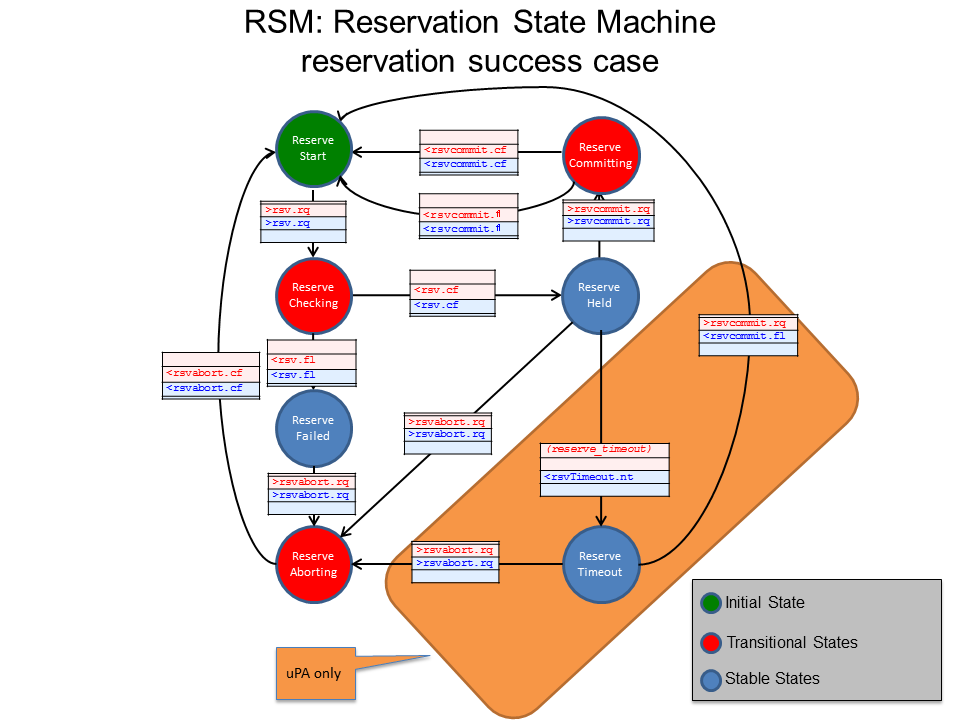


Figure 8: Reservation State Machine

An NSI reservation is created using a two-phase commit process. In the first phase (reserve) the availability of the requested resources is checked; if the resources are available they are held. In the second phase (commit) the requester has the choice to either commit or abort the reservation that was held in the first phase.

If a requester fails to commit a held reservation after a certain period of time the provider times out the reservation and the held resources are released.

Modification of a reservation is supported in NSI CS v2.0. The reserve request message is used for both the initial reservation and subsequent modifications. A version number is specified in the reservation request message. The number is an integer and should be monotonically increasing with each subsequent modification. The version number is updated after a commit results in a transition back to the ReserveStart state. A query will return the currently committed reservation version number. For details of how the version number should be managed is described in Section 5.11.

Modification of start-time, end-time and bandwidth are all supported.

The sequence of operations related to PSM messages MUST conform to the Provision State Machine shown in Figure 8.

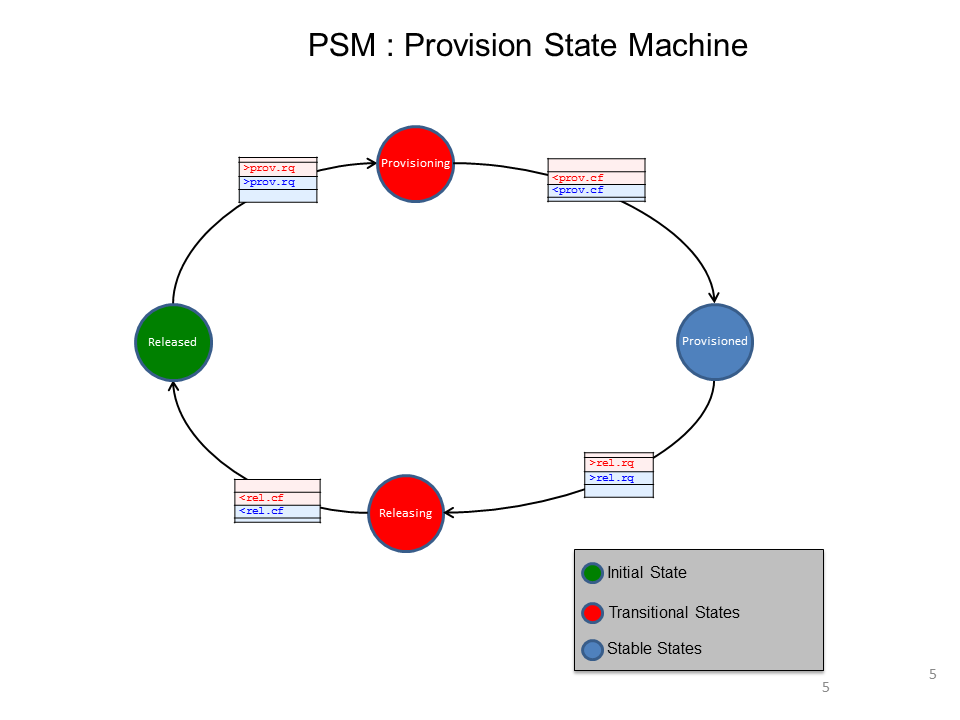


Figure 9: Provision State Machine

The Provision State Machine transits between the Provisioned and the Released stable states, through intermediate transition states. An instance of the PSM is created when an initial reservation is committed, and at that time it starts in the Released state. The PSM transits states independent of the state of the RSM. Note that the transition to the Provisioned state is necessary but on its own is not sufficient to activate the data plane. The Connection in the data plane is active if and only if the PSM is in the Provisioned state AND the start time < current time < end time. See section 4.4 for details of the provisioning and activation.

The PSM is designed to allow a Connection to be repeatedly provisioned and released.

The sequence of operations related to LSM messages MUST conform to the Lifecycle State Machine shown in Figure 10.

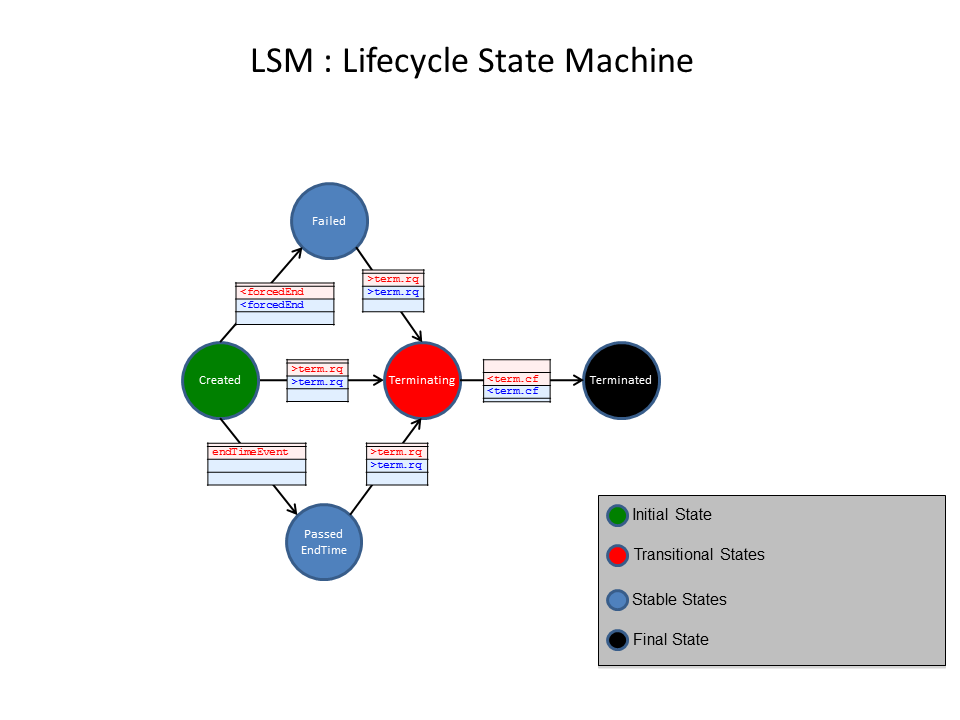


Figure 10: Lifecycle State Machine

The LSM processes the *terminate* and *terminateConfirm* messages. When an *errorEvent* of type *ForcedEnd* is received/sent, the LSM transitions to the Failed state.

## Data Plane Activation

Figure 11 below shows the conditions that MUST be met for data plane activation.

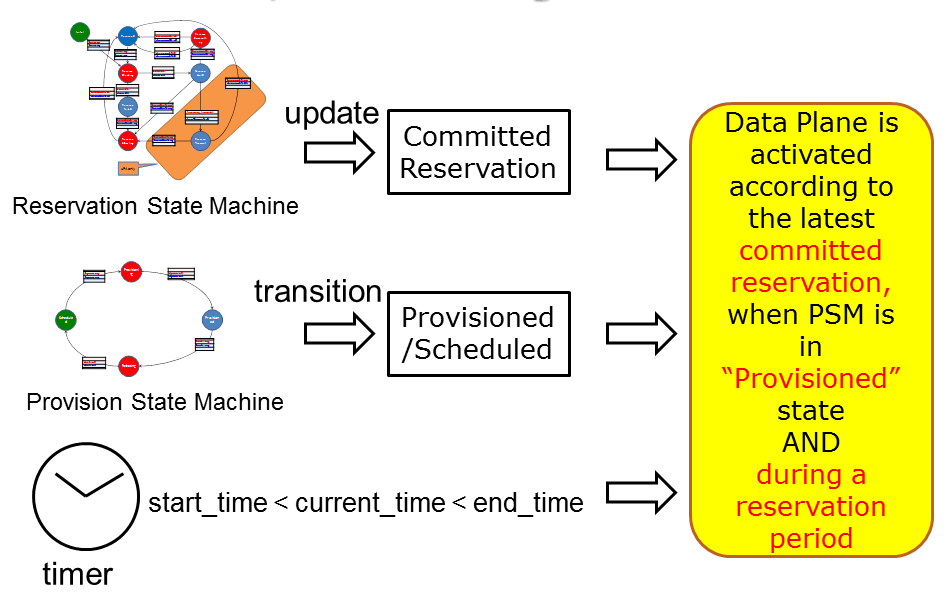


Figure 11: Data Plane activation condition

The Connection can be restored autonomously by the uPA after a failure condition as long as the PSM is in the Provisioned state and time is between *startTime* and *endTime*.

The activation/deactivation of the Data Plane MUST be notified using the *DataPlaneStateChange* notification message. Errors MUST be notified using the generic *errorEvent* message with the following events:

* *activateFailed*: Activation failed at the time when uPA attempted to activate its data plane
* *deactivateFailed*: Deactivation failed at the time when uPA attempted to deactivate its data plane
* *dataplaneError*: On the data plane, the Connection has deactivated unexpectedly. This error condition may be recoverable.
* *forcedEnd*: Something unrecoverable has happened in the uPA/NRM

## Provisioning Sequence

Both automatic and manual provisioning modes MUST be supported. Figure 12 and Figure 13 below. show two examples of how message primitives are used to provision and consequently activate a Connection.

These activation methods can be realized using the semantics described in Figure 11. In the automatic provisioning mode, the provision request message is sent from the RA to the PA before the *startTime*, and the data plane Connection is activated at the *startTime*. If a provision request message is sent after the *startTime*, the data plane Connection is activated when the *provisionRequest* is received by the uPA - this sequence is referred to as manual provisioning. If the uRA wishes to activate the data plane Connection as soon as possible, the uRA should leave the *startTime* blank, which indicates immediate start, and issue a *provisionRequest* message immediately after the reservation is committed. This behavior can be considered as an on-demand mode of provisioning. If the *endTime* is left blank then this is considered to be a request for a permanent Connection.



Figure 12: Automatic Provisioning and Manual Provisioning

A Connection can be repeatedly provisioned and released by provision request messages and release request messages, as shown in Figure 13.



Figure 13: Release and Provisioning

# NSI Process Coordination and Message Transport

## Message Transport

Inherent to the NSI Connection Services is the flexibility to instantiate tree workflows[[1]](#footnote-1) of arbitrary complexity. This flexibility necessitates the formalization of the concept of a Message Transport Layer (MTL). The purpose of the MTL is to present an abstracted message delivery mechanism to the NSI layer. This logical separation of the message delivery from the message themselves aims to simplify the operation of NSI and allow ready migration to new message transport mechanisms.

An MTL that is responsible for end-to-end communications between NSAs MUST be implemented. The MTL has two primary requirements:

* Send and receive messages; The MTL is responsible for encapsulating the NSI message with all the necessary information (e.g. source/destination, port, protocol, etc) for delivery, and removing transport information when a NSI message is received prior to passing it to the Coordinator.
* Verify if a message was received by the intended destination NSA. To do this, the MTL utilizes message receipt acknowledgement and timeouts to determine if a packet was or was not successfully delivered.

NSI messages SHOULD be delivered in the order sent, however, there is no mandatory requirement for the MTL to be reliable or ensure delivery order since this can be accomplished by the higher level processes.

## Message and Process Coordination

As the MTL defines only basic message transport capabilities, the NSA requires more intelligent message and process coordination to function. These capabilities are defined in a logical entity called the coordinator. Even though both the MTL and Coordinator are part of the NSA, the Coordinator is integral to the NSI Stack, whereas the MTL is functionally distinct and can be readily substituted.

The Message coordinator forms a normative part of the NSI CS protocol and MUST be implemented.

The Message coordinator has the following roles:

* To coordinate, track, and aggregate (if necessary) message requests, replies, and notifications
* To process or forward notifications as necessary
* To service query requests

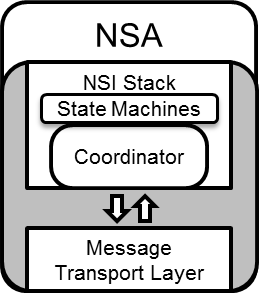


Figure : MTL and Coordinator functions in an NSA

## Communications

Reliable communications is essential to the reliable operation of the NSI. As the MTL provides only basic message transport capabilities, it is the responsibility of the Coordinator to keep track of message states and make decisions accordingly. To do this, the Coordinator MUST maintain the following information on a per NSI request message basis:

* Who the (NSI request) message was sent to
* Was the message received (i.e. ack’ed) or not (i.e. MTL timeout)
* Which NSA has sent back an NSI reply (e.g. \*.confirm, \*.fail, \*.not\_applicable) for the initial NSI request (e.g. \*.request)

## Per Request Information Elements

For each NSI request/reply interaction, the Coordinator maintains several pieces of information that are associated with those messages. This is particularly important for the Aggregator NSAs (AG) which MUST keep track of the message status for each of its children in the request workflow. The information that MUST be retained includes:

* NSA IDs: A list of NSA that the messages were sent to.
* Connection ID: The name that uniquely identifies the connection request/reservation (see “ogf\_nsi\_connection\_types\_v2\_0.xsd” for more detail).
* Correlation ID: The label that identifies messages associated to a unique NSI request/reply interaction. This is used to associate NSI replies to requests, and also to identify messages for re-delivery (i.e. message retries).
* Message status: This provides the message state for each of the NSI requests sent to the various NSAs to reflect the current status, such as; MTL sent, MTL receipt acknowledged, MTL timeout, and Coordinator timeout.

In addition to the detailed information of the status for each child NSA, NSI request (see “request\_segment\_list(Conn\_ID, NSA)” in Figure 16.), the Coordinator MUST also maintain an aggregate message status indicating if the messages were delivered successfully to all the children (see “request\_list(Conn\_ID)” in Figure 16.).

## Timeouts

In order to identify communication failures, both the MTL and Coordinator have defined timeouts to detect breakdowns in certain aspects of the communication channel. The characteristics of these timeouts are outlined below for informational purposes:

* MTL Timeout
  + Symptoms
    - No acknowledgement of message receipt after a pre-determined time period after the message was sent.
  + Causes
    - Failure in end-to-end communication between NSAs.
* Coordinator Timeout
  + Symptoms
    - No NSI reply after a pre-determined time period after the NSI request was sent.
  + Causes
    - Failure in the MTL such that the NSI reply (from the PA) could not be delivered to the RA (the RA).
    - The NSA processing the request (e.g. PA) was unable to reply due to incapacitation.
    - The NSA processing the request (Aggr) was blocked waiting for NSI replies from downstream NSAs. (This scenario can be resolved by adjusting the Coordinator timeout value of the requester.)

As both the MTL and Coordinator timeouts are distinct and can be set exclusively, it is important to understand the interplay between the MTL and Coordinator timeouts in order to mitigate artificial “failures”. The RA may choose to send queries to check the status of a request rather than terminating at timeout.

In the event of an MTL or Coordinator timeout, the Coordinator MUST generate a message delivery failure notification and send it up the workflow tree (towards the uRA).

Timeouts MAY be configurable on a per operation basis and it is suggested that they are set to 2 minutes as a default. Requester side timeouts: It is up to the individual provider to choose appropriate NSA timeouts for their network. As a guide the timeout should be set to 1 minute for reservations to a provider only NSA, and longer for hierarchical requests to aggregator NSAs depending on the number of levels of recursion.

NSA

**Provider Agent**

MH

MTL

MH

NSA

**Requester Agent**

MTL

MH

message

ack

request

return

return

reply

message

reply

ack

notification

message

ack

return

notification

Potential MH Timeout

Potential MTL Timeout

request

Figure 15: Potential MH/MTL timeout sequences

## ACK handling

Delays on the transport layer can result in ACK arriving after the confirm/fail message. The following guidelines are recommended for handling web-service ACKs:

1. For protocol robustness, the NSA should accept any confirm/fail messages even if these are received out-of-order w.r.t. the ACK, i.e. before the associate ACK has been received.
2. The receipt of a confirm/fail message cancels out the need to receive an ACK. So the NSA should not only continue to process the confirm/fail message, but not gate on or wait for the ACK, i.e consequent-messages may be sent without waiting on the receipt of the ACK.
3. As a best practice the NSA SHOULD send the ACK before sending the associated confirm/fail message.
4. The message transport layer takes care of ACK retransmission in case of a packet loss.
5. If the message transport layer is broken, the ACKs will eventually timeout and generate a message transport error that the NSA will need to handle.

## Message transport error handling

Additional error condition handling: The following set of checks is required to pass for messages to be considered vaild and handed on to the relevant state machine, otherwise a message transport layer fault will be returned:

* HTTP authentication – if the message does not have valid credentials it will be rejected with an HTTP 40x message.
* *correlationId* - needed for any acknowledgment, confirmation, or failed message. MUST be unique within the context of the providerNSA otherwise the request cannot be accepted.
* *replyTo* - we will send the confirmation, or failed message back to this location. We do not validate the contents of the endpoint, just that it exists.
* *Reservation* – if the reservation parameters are not present then we reject.
* *requesterNSA* and *providerNSA* – MUST be present and resolve to an *NSnetwork* in topology. Also, the *providerNSA* MUST be the *NSnetwork* that the NSA is managing or the message will be rejected.
* *connectionId* – this is used as the primary reference attribute for Reservation state machines and MUST be present. If the message is for the first *reserve* request then the *connectionId* is left empty and should be completed by the PA.
* If any of these fields are missing or invalid the NSA will return a message transport fault containing the *NSIServiceException* set to an appropriate error message. Typically this will be MISSING\_PARAMETER - "SVC0001", "Invalid or missing parameter" for this generic case and specify attributes identifying the parameter in question.

## Failure Recovery

In NSI CS, there is no inherent expectation that any interim NSAs (i.e not the uRAs) make a decision and take action when it receives a message delivery failure notification. Any Aggregator (AG) that receives the delivery failure notification MUST forward it up the workflow tree. When an AG forwards a notification event up the tree, it SHOULD retain the information concerning the original failure, such as *nsaId*, *connectionId*, and error information. There may be cases where local policy prevents this, in which case the information can be removed or altered.

On receiving the message delivery failure notification, the uRA has two choices:

1. Terminate the reservation; this is done by sending down a *terminate* request through the workflow tree.
2. Request redelivery of the original message; this is done by resending down the original message through the workflow tree.

When the original message is resent down the workflow tree, it will contain the original *correlationId*. AGs receiving the duplicate request should only attempt redelivery of the message to children that it did not receive an acknowledgement for (i.e. MTL timeout) or reply to (i.e. Coordinator timeout) the original message. If the message sent with the original *correlationId* does not match the original message (e.g. different message parameters/content), the message is rejected and an error returned.

## Information maintained by the Coordinator

While per request information (see Section 5.4 Per Request Information Elements) will only persist for the duration of the NSI request/reply interaction, the Coordinator MUST also store information associated with the entire reservation.

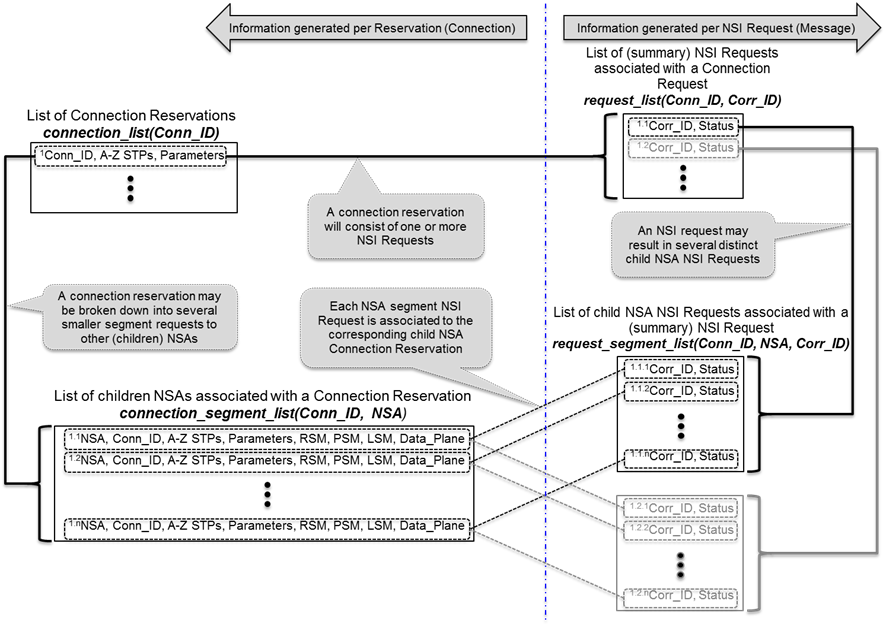


Figure 16: Information maintained by Coordinator for each Connection Reservation and NSI Request

## Per Reservation Information Elements

To support the query function in NSI CS v2.0, an AG Coordinator MUST track the current state (i.e. RSM, PSM, LSM) of all its children as well as the condition of the data plane status. This information is persistent but updated over the lifetime of the reservation (see “connection\_segment\_list(Conn\_ID, NSA)” in Figure 16).

• NSAs: A list of the *nsaId* that are part of the connection request workflow tree.

• Connection IDs: The *connectionId* associated with each NSA in the workflow tree.

• Source and Destination STPs: The *sourceSTP* and *destSTP* of each Connection segment that composes the end-to-end Connection.

• Reservation Parameters: A list of reservation parameters (e.g. *startTime*, *endTime* *bandwidth*, etc) associated with each NSA segment

• RSM States: State of children’s Reservation State Machine and current committed reservation version number

• PSM States: State of children’s Provision State Machine

• LSM States: State of Children’s Lifecycle State Machine

• Data plane states: The status of the children’s data plane (i.e. active/not active), the version of the reservation instantiated in the data plane if it is active (see Section 5.11 Reservation Versioning Information and Section 5.12 Data Plane Status Information for more details), and if the version is consistent.

## Reservation Versioning Information

To support the modification of reservations, the notion of versioning has been introduced to identify the instance of a reservation over its lifetime. Versioning MUST be used as follows:

* Version numbers are integer values ≥ 0 (zero)
* Version numbers are assigned by the RA when a reservation request (i.e. NSI\_rsv.rq) is made to a PA
* An integer ≥ 0 (zero) MAY be assigned by the RA for the initial request, however subsequent modifications to the request MUST use monotonically increasing version numbers (although they need not be sequential)
* If a version number is not specified in an NSI\_rsv.rq, it is assumed to be 0 (zero) regardless if it is the initial or subsequent requests
* An NSI\_rsv.rq with a version number ≤ the (highest) current committed reservation version number will result in a failed request and an appropriate error
* A uPA MUST keep track of
  + Version number of currently committed reservation
  + Version number of pending modification request (if any)
  + Version number of reservation instantiated in the data plane by the NRM
* An Aggregator MUST keep track of
  + Version numbers of currently committed reservations in each child segment
  + Version number of pending modification request (only one modify can be outstanding at any time)
  + Version numbers of reservations instantiated in the data plane in each child segment (see Section 5.12 Data Plane Status Information)
* Version numbers of failed (e.g. timed-out) or aborted modifications are not stored, and therefore can be reused. For example:

1. Successful initial NSI\_req.rq(ver = 2) results in Reservation(v2)
2. Successful modify NSI\_req.rq(ver = 5) results in Reservation(v5)
3. Failed modify NSI\_req.rq(ver = 6) retains Reservation(v5)
4. Subsequent successful modify NSI\_req.rq(ver = 6) results in Reservation(v6)

## Data Plane Status Information

To reflect the state of the data plane, a Coordinator MUST maintain three flags:

* Active (boolean): To indicate whether the data plane is active (in-service or out-of-service)
  + uPA:
    - True => data plane is active
    - False => data plane is not active
  + AG:
    - True => all children’s data planes are active
    - False => one or more children’s data plane is not active
* Version (int): The version of the committed reservation instantiated in the data plane. NB: This field is only valid when “Activate” is true.
  + uPA: Version number of the committed reservation
  + AG: Largest version number of the committed reservation among the children
* VersionConsistent (boolean): Reflects if the “Version” numbers are consistent
  + uPA: This is always True
  + AG:
* True => all children’s “Version” numbers are the same
* False => all children’s “Version” numbers are not the same

When there is a change in the data plane status (i.e. uPA is notified by its NRM, or AG notified by one or more of its children), the Coordinator MUST send up the workflow tree a “DataPlaneStateChange.nt” notification with the updated Activate, Version, and VersionConsistent values.

For the AG, reporting the aggregate data plane state of its children requires some processing. The following pseudo-code describes this behavior:

if all of Children*DataPlaneStatus*[1..n].Active are true then

{

*DataPlaneStatus*.Active = true

}

else {

*DataPlaneStatus*.Active = false

}

*DataPlaneStatus*.Version = maximum(Children*DataPlaneStatus*[1..n].Version)

If all Children*DataPlaneStatus*[1..n].Version are the same, and

all of Children*DataPlaneStatus*[1..n].VersionCosistent are true then

{

*DataPlaneStatus*.VersionConsistent = true

}

else

{

*DataPlaneStatus*.VersionConsistent = false

}

If the new state of an aggregated data plane is the same as the previous aggregated state, the aggregator does not need to send up a *dataPlaneStatus* notification message. In case the aggregated data plane status has changed, the aggregator MUST send up a notification.

# XML Schema Definitions

The NSI CS v2.0 protocol makes use of an XML schema (XSD) to describe the common message header and individual Connection Service operation elements and types. The Web Service Description Language (WSDL) is used to describe the interface or operation bindings, capturing the request, response, and error (fault) interactions. Finally, the WSDL is used to provide a SOAP specific transport binding as a reference specification; however, the XML schema definitions can be utilized to encapsulate the NCI CS protocol into other transport bindings. This section provides a detailed overview of the NSI CS XML schema definitions.

The following namespaces are defined as part of the NSI CS 2.0 protocol:

|  |  |
| --- | --- |
| Description | Namespace URL |
| Common types shared between NSI message header and CS operation definitions. | http://schemas.ogf.org/nsi/2013/04/framework/types |
| NSI message header definition. | http://schemas.ogf.org/nsi/2013/04/framework/headers |
| NSI CS operation specific type definitions. | http://schemas.ogf.org/nsi/2013/04/connection/types |
| NSI CS operation definitions | http://schemas.ogf.org/nsi/2013/04/connection/interface |
| PA interface SOAP binding | http://schemas.ogf.org/nsi/2013/04/connection/provider |
| RA interface SOAP binding | http://schemas.ogf.org/nsi/2013/04/connection/requester |

**Table 4 – PA to RA messages**

## Use of SOAP

The NSI CS utilizes web services as an underlying transport mechanism for protocol messages based on a SOAP over HTTP binding. This section describes the messaging interaction model and standards utilized within an NSI CS implementation.

### Asynchronous Messaging Model

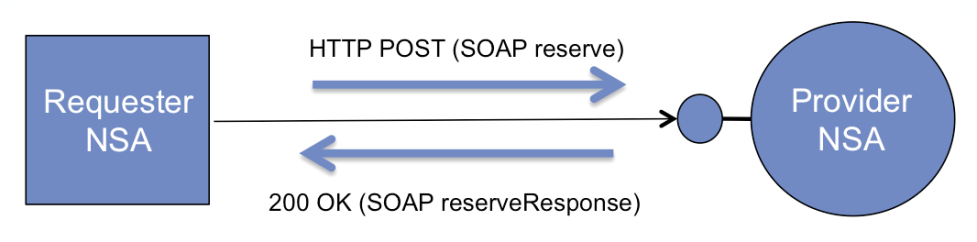
Inherent to the NSI architecture is the need to support long duration operations such as complex reservation requests across multiple domains. This requirement means that a synchronous protocol solution would not suit NSI, consequently the NSI CS is implemented using an asynchronous messaging protocol that allows for indeterminate response times.

The HTTP/SOAP binding as defined in W3C standards is a synchronous request/response interaction model. To help realize the NSI CS as an asynchronous protocol within the context of the synchronous HTTP/SOAP binding, we have defined an asynchronous callback mechanism permitting unblocking of the CS operation request from the CS confirmed and failed response messages.

As an alternative to introducing the complex WS-Addressing specification, we have defined a simple mechanism that permits a RA to provide a *replyTo* URL within the NSI header of the operation request message. This URL is a SOAP endpoint which RA exposes to the PA receive a confirmed or failed message. When the PA has completed processing of the operation request it will invoke the URL provided in the *replyTo* field and deliver the resulting confirmed or failed message to the RA’s SOAP endpoint.

**Figure 17** shows the basic NSI request/response model within the HTTP/SOAP binding. In this case we see the NSI CS *reserve* message being issued from a RA to a PA. The RA will block until either the *reserveResponse* or *ServiceException* is received (notice this operation has a specific response message instead of a generic acknowledgment response). This blocking operation is expected to be extremely short lived as the PA is only acknowledging the acceptance of the request for processing. The following generic behavior can be observed:

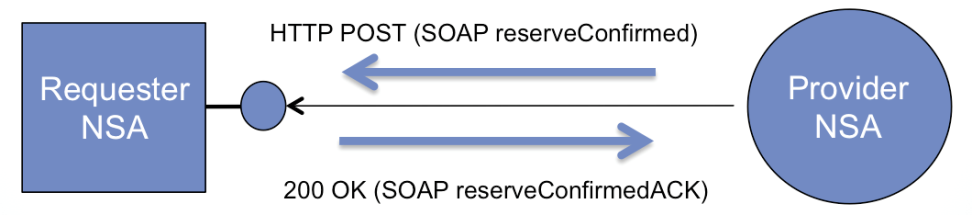
* The HTTP POST request carries the NSI CS operation request with the *replyTo* header element set to the RA’s callback SOAP endpoint.
* The HTTP 200 OK response carries either an acknowledgement or a response to of the NSI CS operation.
* The HTTP socket blocks until the response is returned (synchronous).



**Figure 17 – HTTP/SOAP NSI operation request interaction.**

The PA will maintain the *repyTo* endpoint value specified in the original operation request until it has delivered a confirmed or failed message back to the RA.[[2]](#footnote-2) **Figure 18** illustrates the asynchronous delivery of the *reserveConfirmed* message back to the RA upon success. The following generic behavior can be observed:

* The HTTP POST request carries the NSI CS confirmed request.
* The HTTP 200 OK response carries an acknowledgement indicating successfully delivery of the confirmed message.
* The HTTP socket blocks until the response is returned (synchronous).



**Figure 18 – HTTP/SOAP callback interaction.**

Although most NSA deployments will support the described protocol interactions, there are situations where a RA will not be able to participate in the described HTTP/SOAP asynchronous messaging interaction. An example is where a firewall has been deployed between peering NSA. See Appendix C for a discussion of this firewall issue.

The next section describes NSI CS extensions to support a synchronous messaging model required for RAs that are behind a firewall and are not capable of meeting the public accessibility requirements.

### Synchronous Messaging Model

Version 2.0 of the NSI CS protocol introduced support for synchronous messaging, removing the need for asynchronous callbacks for a requester-only NSA. This is a simple mechanism utilizes the basic CS operation request and query messages to provide a functional polling solution removing the need for asynchronous callbacks. This has been added specifically to help address the firewall issue described in the appendix.

The synchronous messaging model relies on four new mechanisms to remove the need for asynchronous callbacks, and permit a firewall safe RA implementation:

1. The RA informs the PA that it is not interested in receiving asynchronous callbacks by not specifying a *replyTo* address in the NSI header of the CS operation request. The PA will perform the requested operation, but will not send the confirmed/failed message back to the RA.
2. The *reserve* operation returns the PA allocated *connectionId* for the reservation in the synchronous *reserveResponse* message.
3. The RA can use the *querySummarySync* operation to synchronously retrieve reservation information based on the *connectionId*, monitoring the state machine transitions to determine progress and result of operation.
4. Notifications generated against a *connectionId* are identified in the reservation query result, and can be retrieved using the *queryNotificationSync* operation.

### Standard Compliance

The NSI CS protocol is specified using WSDL 1.1 and utilizes the SOAP 1.1 message encoding as identified by the namespaces:

* soap - "http://schemas.xmlsoap.org/soap/envelope/"
* xsi - "http://www.w3.org/2001/XMLSchema-instance"
* xsd - "http://www.w3.org/2001/XMLSchema"
* soapenc - "http://schemas.xmlsoap.org/soap/encoding/"
* wsdl - "http://schemas.xmlsoap.org/wsdl/"
* soapbind - "http://schemas.xmlsoap.org/wsdl/soap/"

The specific NSI CS operation being invoked is identified using the “Soapaction:” element in the HTTP header as per section 6.1.1 of “Simple Object Access Protocol (SOAP) 1.1” found at http://www.w3.org/TR/SOAP. This allows for better compatibility between SOAP implementations even though it is not explicitly required as per WS-I Basic Profile 1.1 http://www.ws-i.org/Profiles/BasicProfile-1.1-2006-04-10.html. The NSI header as defined with XML schema does not contain a specific “operation” element as this is included in the “Soapaction:” element and would be duplicate data.

## NSI CS Versioning

The common way of version SOAP and XSD is by using XML namespaces. Each of the WSDL and XSD schema files defined as part of the NSI CS protocol are identified through their designated namespace URL (for example, <http://schemas.ogf.org/nsi/2013/04/framework/headers> for the NSI framework header definition). This versioning mechanism is vital for ensuring end-to-end syntax consistency for message exchange; however, these namespaces do not identify specific behavioral aspects of the protocol. To solve this NSI v2.0 has introduced a protocol version field within the NSI header to convey both the syntactic and behavior version of the protocol. This allows additional versions to be defined that can change behavior aspects without upgrading the base WSDL or XSD definitions.

Versioning of the NSI CS protocol utilizes Internet Assigned Numbers Authority (IANA) MIME Media Types as a standard mechanism for distinguishing between releases of the protocol.

Table **5** below enumerates the MIME Media Types defined for each version of the protocol, and the specific protocol interface role the NSA supports. These are the string values that will be populated in the *protocolVersion* field of the NSI header for each message sent (see section 6.3).

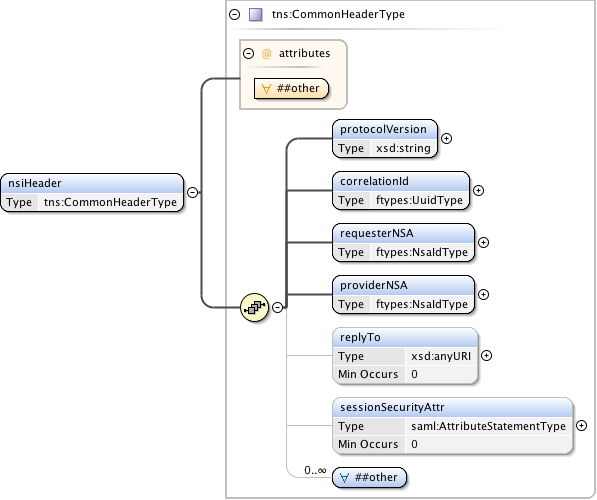
|  |  |  |
| --- | --- | --- |
| Version | Interface | MIME Media Type |
| NSI CS version 1.0 | Provider | “application/vdn.ogf.nsi.cs.v1.provider+soap” |
| NSI CS version 1.0 | Requester | “application/vdn.ogf.nsi.cs.v1.requester+soap” |
| NSI CS version 1.1 | Provider | “application/vdn.ogf.nsi.cs.v1-1.provider+soap” |
| NSI CS version 1.1 | Requester | “application/vdn.ogf.nsi.cs.v1-1.requester+soap” |
| NSI CS version 1.0 | Provider | “application/vdn.ogf.nsi.cs.v2.provider+soap” |
| NSI CS version 1.0 | Requester | “application/vdn.ogf.nsi.cs.v2.requester+soap” |

**Table 5 – NSI CS protocol version MIME Media Types.**

## nsiHeader element

***Namespace definition:*** http://schemas.ogf.org/nsi/2013/04/framework/headers

The *nsiHeader* element contains attributes common to all NSI CS operations, and therefore, is sent as part of every NSI CS message exchange. Attributes included in the header provide protocol versioning, basic message routing for the protocol, and user security infrastructure. For the SOAP protocol binding, the *nsiHeader* element is encapsulated in the SOAP header, while the NSI specific operation is encapsulated in the SOAP body.



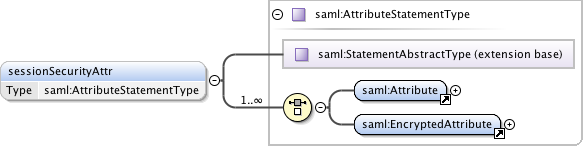
**Figure 19 – *nsiHeader* structure.**

***Parameters***

The *nsiHeader* has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *protocolVersion* | M | A string identifying the specific protocol version carried in this NSI message. The protocol version is modeled separately from the namespace of the WSDL and XML schema to capture behavioral changes that cannot be modeled in schema definition, and to avoid updating of the schema namespace. |
| *correlationId* | M | An identifier provided by the RA used to correlate to an asynchronous response from the responder. It is recommended that a Universally Unique Identifier (UUID) URN as per IETF RFC 4122 be used as a globally unique value. |
| *requesterNSA* | M | The NSA identifier for the NSA acting in the RA role for the specific NSI operation. |
| *providerNSA* | M | The NSA identifier for the NSA acting in the PA role for the specific NSI operation. |
| *replyTo* | O | The RA's SOAP endpoint address to which asynchronous messages associated with this operation request will be delivered. This is only populated for the original operation request (*reserve*, *provision*, *release*, *terminate*, and the query messages), and not for any additional messaging associated with the operation. If no endpoint value is provided in an operation request, then it is assumed the RA is not interested in a response and will use alternative mechanism to determine the result (i.e. polling using query). |
| *sessionSecurityAttributes* | O | Security attributes associated with the end user's NSI session. This field can be used to perform authentication, authorization, and policy enforcement of end user requests. It is only provided in the operation request (*reserve*, *provision*, *release*, *terminate*, and the query messages), and not for any additional messaging associated with the operation. |
| *any element and anyAttribute* | O | Provides a flexible mechanism allowing additional elements in the protocol header for exchange between two-peered NSA. Use of this element field is beyond the current scope of this NSI specification, but may be used in the future to extend the existing protocol without requiring a schema change. Additionally, the field can be used between peered NSA to provide additional context not covered in the existing specification, however, this is left up to specific peering agreements. |

**Table 6 *nsiHeader* parameters**



**Figure 20 – *sessionSecurityAttr* structure.**

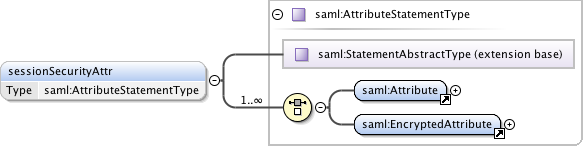
The following table describes each message and it’s use of the individual header parameters. The “Soapaction” parameter identified in the last column of the table is carried in the HTTP request attributes and not the NSI specific header.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | |  | *Header parameters* | | | | | | | |
| M = Mandatory  O = Optional  N/A = Not Applicable | | | *protocolVersion* | *correlationId* | *requesterNSA* | *providerNSA* | *replyTo* | *sessionSecurityAttributes* | *other* | *Soapaction* |
|  | | *reserve* | M | M | M | M | O | O | O | M |
|  | | *reserveResponse* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *reserveConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *reserveConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *reserveFailed* | M | M | M | M | N/A | N/A | O | M |
|  | | *reserveFailedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *reserveCommit* | M | M | M | M | O | O | O | M |
|  | | *reserveCommitACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *reserveCommitConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *reserveCommitConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *reserveCommitFailed* | M | M | M | M | N/A | N/A | O | M |
|  | | *reserveCommitFailedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *reserveAbort* | M | M | M | M | O | O | O | M |
|  | | *reserveAbortACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *reserveAbortConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *reserveAbortConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *provision* | M | M | M | M | O | O | O | M |
|  | | *provisionACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *provisionConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *provisionConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
| Messaging  Primitives | |  |  |  |  |  |  |  |  |  |
| *release* | M | M | M | M | O | O | O | M |
| *releaseACK* | M | M | M | M | N/A | N/A | O | N/A |
| *releaseConfirmed* | M | M | M | M | N/A | N/A | O | M |
| *releaseConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  |  |  |  |  |  |  |  |  |
|  | | *terminate* | M | M | M | M | O | O | O | M |
|  | | *terminateACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *terminateConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *terminateConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *querySummary* | M | M | M | M | M | O | O | M |
|  | | *querySummaryACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *querySummaryConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *querySummaryConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *querySummaryFailed* | M | M | M | M | N/A | N/A | O | M |
|  | | *querySummaryFailedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *queryRecursive* | M | M | M | M | M | O | O | M |
|  | | *queryRecursiveACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *queryRecursiveConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *queryRecursiveConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *queryRecursiveFailed* | M | M | M | M | N/A | N/A | O | M |
|  | | *queryRecursiveFailedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *querySummarySync* | M | M | M | M | N/A | O | O | M |
|  | | *querySummarySyncConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *querySummarySyncFailed* | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *errorEvent* | M | M | M | M | N/A | N/A | O | M |
|  | | *errorEventACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *reserveTimeout* | M | M | M | M | N/A | N/A | O | M |
|  | | *reserveTimeoutACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *dataPlaneStateChange* | M | M | M | M | N/A | N/A | O | M |
|  | | *dataPlaneStateChangeACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *messageDeliveryTimeout* | M | M | M | M | N/A | N/A | O | M |
|  | | *messageDeliveryTimeoutACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *queryNotification* | M | M | M | M | M | O | O | M |
|  | | *queryNotificationACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *queryNotificationConfirmed* | M | M | M | M | N/A | N/A | O | M |
|  | | *queryNotificationConfirmedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | | *queryNotificationFailed* | M | M | M | M | N/A | N/A | O | M |
|  | | *queryNotificationFailedACK* | M | M | M | M | N/A | N/A | O | N/A |
|  | |  |  |  |  |  |  |  |  |  |
|  | | *queryNotificationSync* | M | M | M | M | N/A | O | O | M |
|  | | *queryNotificationSyncConfimed* | M | M | M | M | N/A | N/A | O | M |
|  | | *queryNotificationSyncFailed* | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

**Table 7 – NSI CS message use of header fields**

### sessionSecurityAttributes type

The *sessionSecurityAttributes* element is defined using a standardized SAML *AtttributeStatementType* imported from the SAML namespace *urn:oasis:names:tc:SAML:2.0:assertion*.



**Figure 21 – *sessionSecurityAttributes* type.**

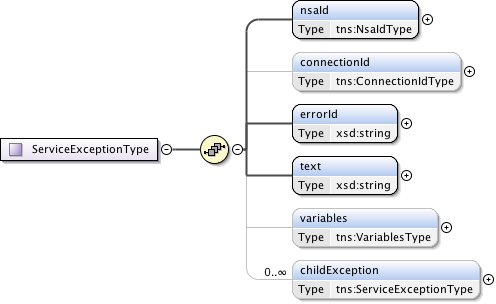
## Common types

***Namespace definition:*** http://schemas.ogf.org/nsi/2013/04/framework/types

These are the common types shared between NSI message header and CS operation definitions.

### ServiceExceptionType

Common service exception used for SOAP faults and operation failed messages.



**Figure 22 – *ServiceExceptionType* type.**

***Parameters***

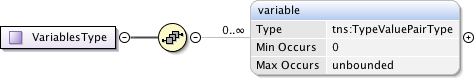
The *ServiceExceptionType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *nsaId* | M | NSA that generated the service exception. |
| *connectionId* | O | The *connectionId* associated with the reservation impacted by this error. |
| *errorId* | M | Error identifier uniquely identifying each known fault within the protocol. |
| *text* | M | User-friendly message text describing the error. |
| *providerNSA* | M | The NSA identifier for the NSA acting in the PA role for the specific NSI operation. |
| *variables* | O | An optional collection of type/value pairs providing additional information relating to the error. |
| *childException* | O | Hierarchical list of service exceptions capturing failures within the request tree. |

**Table 8 – *ServcieExceptionType* parameters**.

### VariablesType

A type definition providing a set of zero or more type/value variables used for modeling generic attributes.



**Figure 23 – *NsaIdType* type.**

***Parameters***

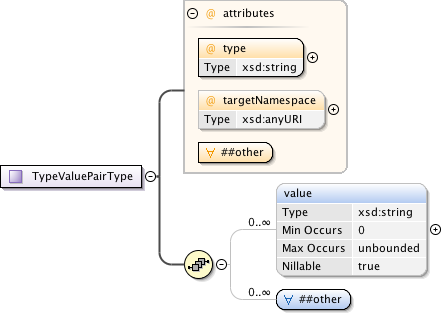
The *VariablesType* has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *variable* | O | The variable containing the type/values. |

**Table 9 – *VariablesType* parameters.**

### TypeValuePairType

*TypeValuePairType* is a simple type and multi-value tuple. Includes simple string type and value, as well as more advanced extensions if needed. A targetNamespace attribute is included to provide additional context where needed.



**Figure 24 – *TypeValuePairType* type.**

***Parameters***

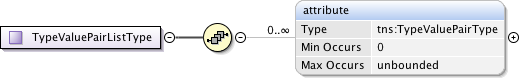
The *TypeValuePairType* has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *type* | M | A string representing the name of the type. |
| *targetNamespace* | O | An optional URL to qualify the name space of the capability. |
| *anyAttribute* |  | Provides a flexible mechanism allowing additional attributes non-specified to be provided as needed for peer-to-peer NSA communications. Use of this attribute field is beyond the current scope of this NSI specification, but may be used in the future to extend the existing protocol without requiring a schema change. |
| *value* | O | A string value corresponding to type. |
| *any* | O | Provides a flexible mechanism allowing additional elements to be provided as an alternative, or in combination with value. Use of this element field is beyond the current scope of this NSI specification, but may be used in the future to extend the existing protocol without requiring a schema change. |

**Table 10 – *TypeValuePairType*** **parameters**

### TypeValuePairListType

A simple holder type providing a list definition for the attribute type/values structure.



**Figure 25 – *TypeValuePairListType* type.**

***Parameters***

The *TypeValuePairListType* has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *attribute* | O | An instance of a type/value structure. |

**Table 11 – *TypeValuePairListType*** **parameters**

### ConnectionIdType

A *connectionId* is a simple string value that uniquely identifies a reservation segment within the context of a PA. This value is not globally unique.



**Figure 26 – *ConnectionIdType* type.**

### DateTimeType

The time zone support of W3C XML Schema is quite controversial and needs some additional constraints to avoid comparison problems. These patterns can be kept relatively simple since the syntax of the datetime is already checked by the schema validator and only simple additional checks need to be added. This type definition checks that the time part ends with a "Z" or contains a sign. Values MUST correspond to the following pattern ".+T.+(Z|[+-].+)"



**Figure 27 – *TypeValuePairListType* type.**

### NsaIdType

*NsaIdType* is a specific type for a Network Services Agent (NSA) identifier that is populated with a OGF URN (reference artifact 6478 "Procedure for Registration of Subnamespace Identifiers in the URN:OGF Hierarchy") to be used for compatibility with other external systems.



**Figure 28 – *NsaIdType* type.**

### UuidType

Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 | ISO/IEC 9834-8:2005 and IETF RFC 4122. Values MUST correspond to the following pattern “urn:uuid:[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}”.



**Figure 29 – *TypeValuePairListType* type.**

## NSI CS operation specific type definitions.

***Namespace definition:*** <http://schemas.ogf.org/nsi/2013/04/connection/types>

These are the NSI CS specific operations element definitions for each message defined in the protocol.

### Reserve message elements

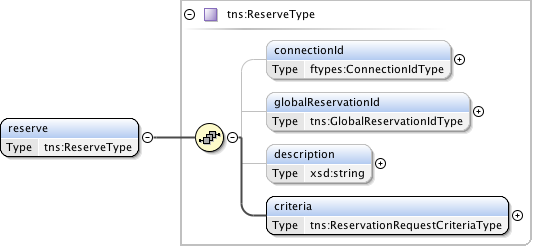
The *reserve* message is sent from a RA to a PA when a new reservation is being requested, or a modification to an existing reservation is required. The *reserveResponse* indicates that the PA has accepted the reservation request for processing and has assigned it the returned *connectionId*. A *reserveConfirmed* or *reserveFailed* message will be sent asynchronously to the RA when reserve operation has completed processing.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *reserve* | *reserveResponse* | *serviceException* |
| Confirmed | PA to RA | *reserveConfirmed* | *reserveConfirmedACK* | *serviceException* |
| Failed | PA to RA | *reserveFailed* | *reserveFailedACK* | *serviceException* |

**Table 12 *Reserve* message elements**

#### Request: reserve

The NSI CS *reserve* message allows a RA to reserve network resources for a Connection between two STP's within the Network constrained by the provided service parameters. This *reserve* message allows a RA to check the feasibility of a connection reservation, or modification an existing connection reservation. Any resources associated with the reservation or modification operation will be allocated and held until a *reserveCommit* message is received for the reservation or timeout occurs (whichever arrives first).



**Figure 30 – *reserve* request message structure.**

***Parameters***

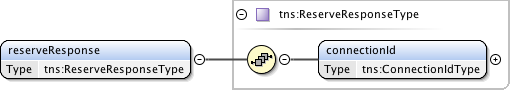
The *reserve*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. Provided in reserve request only when an existing reservation is being modified. This MAY be populated with a Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 |ISO/IEC 9834-8:2005 and IETF RFC 4122. |
| *globalReservationId* | An optional global reservation id that can be used to correlate individual related service reservations through the network. This MUST be populated with a Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 |ISO/IEC 9834-8:2005 and IETF RFC 4122. |
| *description* | An optional description for the service reservation. |
| *criteria* | Reservation request criteria including start and end time, service attributes, and requested path for the service. |

**Table 13 *Reserve* message parameters**

***Response***

If the *reserve*operation is successful, a *reserveResponse* message is returned, otherwise a *serviceException* is returned. A PA sends this *reserveResponse* message immediately after receiving the reservation request to inform the RA of the *connectionId* allocated to their reservation request. This *connectionId* can then be used to query reservation progress.



**Figure 31 – *reserveResponse* message structure.**

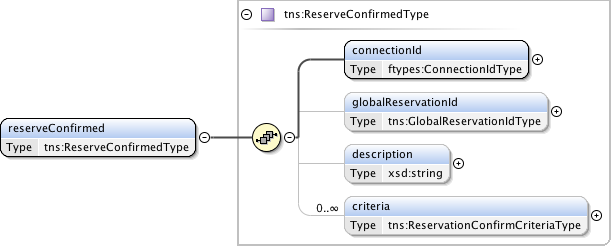
The *reserveResponse* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation request. This value will be unique within the context of the PA. |

**Table 14 *reserveResponse*** **message parameters**

#### Confirmation: reserveConfirmed

A PA sends this positive *reserveConfirmed* response message to the RA that issued the original reserve request message. Receipt of this message is an indication that the requested reservation parameters were available and will be held until a *reserveCommit* message is received for the reservation or timeout occurs (whichever arrives first).



**Figure 32 – *reserveConfirmed* message structure.**

***Parameters***

The *reserveConfirmed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. Provided in reserve request only when an existing reservation is being modified. |
| *globalReservationId* | An optional global reservation id that can be used to correlate individual related service reservations through the network. This MUST be populated with a Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 |ISO/IEC 9834-8:2005 and IETF RFC 4122. |
| *description* | An optional description for the service reservation. |
| *criteria* | A set of versioned and confirmed reservation criteria information including start and end time, service attributes, and requested path for the service. |

**Table 15 *reserveConfirmed* message parameters**

***Response***

If the *reserveConfirmed*operation is successful, a *reserveConfirmedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *reserveConfirmedACK*message immediately after receiving the *reserveConfirmed*request to acknowledge to the PA the *reserveConfirmed*request has been accepted for processing. The *reserveConfirmedACK*message is implemented using the generic acknowledgement message.

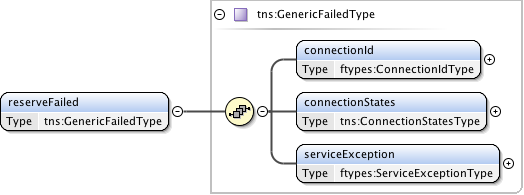


**Figure 33 – *reserveConfirmedACK*** **message structure.**

The *reserveConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Failed: reserveFailed

A PA sends this negative ***reserveFailed*** response to the RA that issued the original reservation request message if the requested reservation criteria could not be met. This message is also sent in response to a reserve request for a modification to an existing schedule if the required modification is not possible.



**Figure 34 – *reserveFailed* message structure.**

***Parameters***

The *reserveFailed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. |
| *connectionStates* | Overall connection state for the reservation. |
| *serviceException* | Specific error condition indicating the reason for the failure. |

**Table 16 *reserveFailed*** **message parameters**

***Response***

If the *reserveFailed* operation is successful, a *reserveFailedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *reserveFailedACK*message immediately after receiving the *reserveFailed* request to acknowledge to the PA the *reserveFailed* request has been accepted for processing. The *reserveFailedACK*message is implemented using the generic acknowledgement message.



**Figure 35 – *reserveFailedACK*** **message structure.**

The *reserveFailedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### reserveCommit message elements

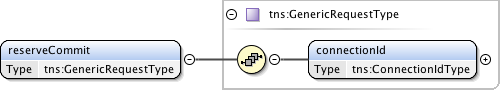
The *reserveCommit* message is sent from a RA to a PA when a reservation or modification to an existing reservation is being committed. This reservation MUST currently reside in the Reserve Held state for this operation to be accepted. The *reserveCommitACK* indicates that the PA has accepted the modify request for processing. A *reserveCommitConfirmed* or *reserveCommitFailed* message will be sent asynchronously to the RA when reserve or modify processing has completed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *reserveCommit* | *reserveCommitACK* | *serviceException* |
| Confirmed | PA to RA | *reserveCommitConfirmed* | *reserveCommitConfirmedACK* | *serviceException* |
| Failed | PA to RA | *reserveCommitFailed* | *reserveCommitFailedACK* | *serviceException* |

**Table 17 *ReserveCommit* message elements**

#### Request: reserveCommit

The NSI CS *reserveCommit*message allows a RA to commit a previously allocated reservation or modification on a reservation. The *reserveCommit* request MUST arrive at the Provider Agent before the reservation timeout occurs.



**Figure 36 – *reserveCommit* request message structure.**

***Parameters***

The *reserveCommit*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for the reservation that is to be committed. |

**Table 18 *reserveCommit* message parameters**

***Response***

If the *reserveCommit*operation is successful, a *reserveCommitACK*message is returned, otherwise a *serviceException* is returned. A PA sends this *reserveCommitACK*message immediately after receiving the *reserveCommit*request to acknowledge to the RA the *reserveCommit*request has been accepted for processing. The *reserveCommitACK*message is implemented using the generic acknowledgement message.

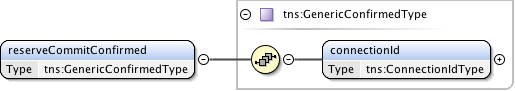


**Figure 37 – *reserveCommitACK* message structure.**

The *reserveCommitACK* message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: reserveCommitConfirmed

This *reserveCommitConfirmed* message is sent from a PA to RA as an indication of a successful *reserveCommit* request for a reservation previously in a Reserve Held state.



**Figure 38 – *reserveCommitConfirmed* message structure.**

***Parameters***

The *reserveCommitConfirmed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The connection identifier for the reservation that was committed. |

**Table 19 *reserveCommitConfirmed* message parameters**

***Response***

If the *reserveCommitConfirmed*operation is successful, a *reserveCommitConfirmedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *reserveCommitConfirmedACK*message immediately after receiving the *reserveCommitConfirmed*request to acknowledge to the PA the *reserveCommitConfirmed*request has been accepted for processing. The *reserveCommitConfirmedACK*message is implemented using the generic acknowledgement message.

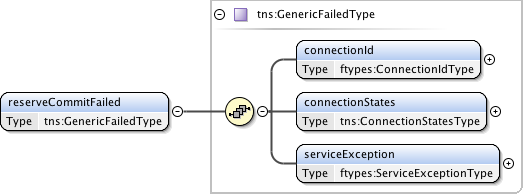


**Figure 39 – *reserveAbortConfirmedACK*** **message structure.**

The *reserveCommitConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Failed: reserveCommitFailed

This *reserveCommitFailed* message is sent from a PA to RA as an indication of a *reserve* (or modify) commit failure. This is in response to an original *reserveCommit* request from the associated RA.



**Figure 40 – *reserveCommitFailed* message structure.**

***Parameters***

The *reserveCommitFailed*message takes the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. |
| *connectionStates* | Overall connection state for the reservation. |
| *serviceException* | Specific error condition indicating the reason for the failure. |

**Table 20 *reserveCommitFailed* message parameters**

***Response***

If the *reserveCommitFailed* operation is successful, a *reserveCommitFailedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *reserveCommitFailedACK*message immediately after receiving the *reserveCommitFailed* request to acknowledge to the PA the *reserveCommitFailed* request has been accepted for processing. The *reserveCommitFailedACK*message is implemented using the generic acknowledgement message.



**Figure 41 – *reserveCommitFailedACK*** **message structure.**

The *reserveCommitFailedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### reserveAbort message elements

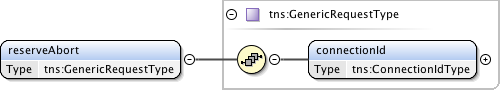
The *reserveAbort*message is sent from a RA to a PA when an initial reservation request, or modification to an existing reservation is to be aborted, and the reservation state machine returned to the previous version of the reservation. The *reserveAbortACK*indicates that the PA has accepted the abort request for processing. A *reserveAbortConfirmed*message will be sent asynchronously to the RA when the abort processing has completed. There is no associated Failed message for this operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *reserveAbort* | *reserveAbortACK* | *serviceException* |
| Confirmed | PA to RA | *reserveAbortConfirmed* | *reserveAbortConfirmedACK* | *serviceException* |
| Failed | N/A | N/A | N/A | N/A |

**Table 21 *reserveCommitFailed* message elements**

### Request: reserveAbort

The NSI CS *reserveAbort*message allows a RA to abort a previously requested reservation or modification on a reservation.



**Figure 42 – *reserveAbort* request message structure.**

***Parameters***

The *reserveAbort*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for the reservation or modification that is to be aborted. |

**Table 22 *reserveAbort* message parameters**

***Response***

If the *reserveAbort*operation is successful, a *reserveAbortACK*message is returned, otherwise a *serviceException* is returned. A PA sends this *reserveAbortACK*message immediately after receiving the *reserveAbort*request to acknowledge to the RA the *reserveAbort*request has been accepted for processing. The *reserveAbortACK*message is implemented using the generic acknowledgement message.

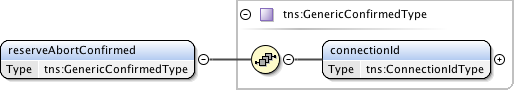


**Figure 43 – *reserveAbortACK*** **message structure.**

The *reserveAbortACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: reserveAbortConfirmed

This *reserveAbortConfirmed*message is sent from a PA to RA as an indication of a successful *reserveAbort*request. The reservation in question will have any pending modifications cancelled and returned to the reservation state existing before the modification.



**Figure 44 – *reserveAbortConfirmed*** **message structure.**

***Parameters***

The *reserveAbortConfirmed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The connection identifier for the reservation that was aborted. |

**Table 23 *reserveAbortConfirmed* message parameters**

***Response***

If the *reserveAbortConfirmed*operation is successful, a *reserveAbortConfirmedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *reserveAbortConfirmedACK*message immediately after receiving the *reserveAbortConfirmed*request to acknowledge to the PA the *reserveAbortConfirmed*request has been accepted for processing. The *reserveAbortConfirmedACK*message is implemented using the generic acknowledgement message.



**Figure 45 – *reserveAbortConfirmedACK*** **message structure.**

The *reserveAbortConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Provision message elements

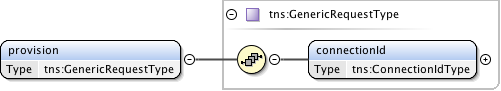
The *provision* message is sent from a RA to a PA when an existing reservation is to be transitioned into a provisioned state. The *provisionACK* indicates that the PA has accepted the *provision* request for processing. A *provisionConfirmed* or message will be sent asynchronously to the RA when *provision* processing has completed. There is no associated Failed message for this operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *provision* | *provisionACK* | *serviceException* |
| Confirmed | PA to RA | *provisionConfirmed* | *provisionConfirmedACK* | *serviceException* |
| Failed | N/A | N/A | N/A | N/A |

**Table 24 Provision message elements**

#### Request: provision

The NSI CS *provision* message allows a RA to transition a previously requested reservation into a provisioned state. A reservation in a provisioned state will activate associated data plane resources during the scheduled reservation time.



**Figure 46 – provision request message structure.**

***Parameters***

The *provision* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for the reservation to be provisioned. |

**Table 25 Provision message parameters**

***Response***

If the *provision* operation is successful, a *provisionACK*message is returned, otherwise a *serviceException* is returned. A PA sends this *provisionACK*message immediately after receiving the *provision*request to acknowledge to the RA the *provision*request has been accepted for processing. The *provisionACK*message is implemented using the generic acknowledgement message.

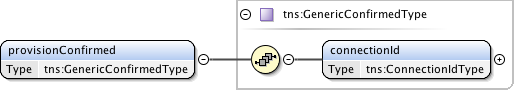


**Figure 47 – *provisionACK*** **message structure.**

The *provisionACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: provisionConfirmed

This *provisionConfirmed*message is sent from a PA to RA as an indication of a successful *provision*request. This is in response to an original *provision* request from the associated RA.



**Figure 48 – *provisionConfirmed*** **message structure.**

***Parameters***

The *provisionConfirmed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The connection identifier for the reservation that was provisioned. |

**Table 26 *provisionConfirmed* message parameters**

***Response***

If the *provisionConfirmed*operation is successful, a *provisionConfirmedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *provisionConfirmedACK*message immediately after receiving the *provisionConfirmed*request to acknowledge to the PA the *provisionConfirmed*request has been accepted for processing. The *provisionConfirmedACK*message is implemented using the generic acknowledgement message.



**Figure 49 – *provisionConfirmedACK*** **message structure.**

The *provisionConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### release message elements

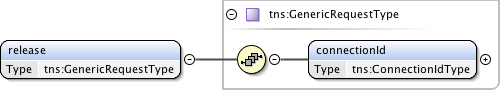
The *release* message is sent from a RA to a PA when an existing reservation is to be transitioned into a Released state. The *releaseACK* indicates that the PA has accepted the release request for processing. A *releaseConfirmed* message will be sent asynchronously to the RA when release processing has completed. There is no associated failed message for this operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *release* | *releaseACK* | *serviceException* |
| Confirmed | PA to RA | *releaseConfirmed* | *releaseConfirmedACK* | *serviceException* |
| Failed | N/A | N/A | N/A | N/A |

**Table 27 Release message elements**

#### Request: release

The NSI CS *release* message allows a RA to transition a previously requested reservation into a released state. A reservation in a released state will deactivate associated data plane resources.



**Figure 50 – *release* request message structure.**

***Parameters***

The *release* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for the reservation to be released. |

**Table 28 Release message parameters**

***Response***

If the *release* operation is successful, a *releaseACK*message is returned, otherwise a *serviceException* is returned. A PA sends this *releaseACK*message immediately after receiving the *release*request to acknowledge to the RA the *release*request has been accepted for processing. The *releaseACK*message is implemented using the generic acknowledgement message.

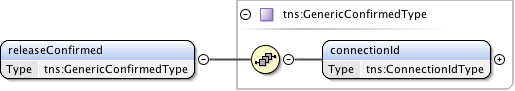


**Figure 51 – *releaseACK*** **message structure.**

The *releaseACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: releaseConfirmed

This *releaseConfirmed*message is sent from a PA to RA as an indication of a successful *release*request. This is in response to an original *release* request from the associated RA.



**Figure 52 – *releaseConfirmed*** **message structure.**

*Parameters*

The *releaseConfirmed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The connection identifier for the reservation that was released. |

**Table 29 *releaseConfirmed* message parameters**

***Response***

If the *releaseConfirmed*operation is successful, a *releaseConfirmedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *releaseConfirmedACK*message immediately after receiving the *releaseConfirmed*request to acknowledge to the PA the *releaseConfirmed*request has been accepted for processing. The *releaseConfirmedACK*message is implemented using the generic acknowledgement message.



**Figure 53 – *releaseConfirmedACK*** **message structure.**

The *releaseConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### terminate message elements

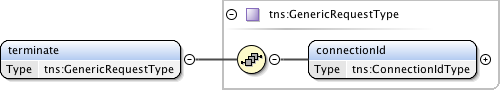
The *terminate* message is sent from a RA to a PA when an existing reservation is to be transitioned into a terminated state and all associated resources in the network are freed. The *terminateACK* indicates that the PA has accepted the *terminate* request for processing. A *terminateConfirmed* message will be sent asynchronously to the RA when *terminate* processing has completed. There is no associated Failed message for this operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *terminate* | *terminateACK* | *serviceException* |
| Confirmed | PA to RA | *terminateConfirmed* | *terminateConfirmedACK* | *serviceException* |
| Failed | N/A | N/A | N/A | N/A |

**Table 30 terminate** **message elements**

#### Request: terminate

The NSI CS *terminate* message allows a RA to transition a previously requested reservation into a Terminated state. A reservation in a Terminated state will release associated resources.



**Figure 54 – *terminate* request message structure.**

***Parameters***

The *terminate* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for the reservation to be terminated. |

**Table 31 *terminate*** **message parameters**

***Response***

If the *terminate* operation is successful, a *terminateACK*message is returned, otherwise a *serviceException* is returned. A PA sends this *terminateACK*message immediately after receiving the *terminate*request to acknowledge to the RA the *terminate*request has been accepted for processing. The *terminateACK*message is implemented using the generic acknowledgement message.

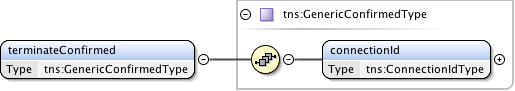


**Figure 55 – *terminateACK*** **message structure.**

The *terminateACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: terminateConfirmed

This *terminateConfirmed*message is sent from a PA to RA as an indication of a successful *terminate*request. This is in response to an original *terminate* request from the associated RA.



**Figure 56 – *terminateConfirmed*** **message structure.**

***Parameters***

The *terminateConfirmed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The connection identifier for the reservation that was terminated. |

**Table 32 *terminateConfirmed* message parameters**

***Response***

If the *terminateConfirmed*operation is successful, a *terminateConfirmedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *terminateConfirmedACK*message immediately after receiving the *terminateConfirmed*request to acknowledge to the PA the *terminateConfirmed*request has been accepted for processing. The *terminateConfirmedACK*message is implemented using the generic acknowledgement message.



**Figure 57 – *terminateConfirmedACK*** **message structure.**

The *terminateConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### querySummary message elements

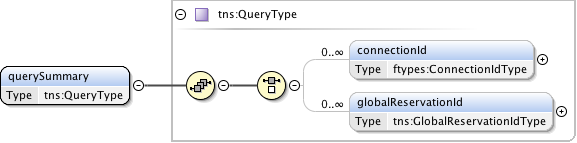
The *querySummary* message is sent from a RA to a PA to determine the status of existing reservations. The *querySummaryACK* indicates that the PA has accepted the *querySummary* request for processing. A *querySummaryConfirmed* or *querySummaryFailed* message will be sent asynchronously to the RA when *querySummary* processing has completed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *querySummary* | *querySummaryACK* | *serviceException* |
| Confirmed | PA to RA | *querySummaryConfirmed* | *querySummaryConfirmedACK* | *serviceException* |
| Failed | PA to RA | *querySummaryFailed* | *querySummaryFailedACK* | *serviceException* |

**Table 33 *querySummary*** **message elements**

#### Request: querySummary

The *querySummary* message provides a mechanism for a RA to query the PA for a set of connection service reservation instances between the RA-PA pair. This message can be used to monitor the progress of a reservation.  
   
Elements compose a filter for specifying the reservations to return in response to the *querySummary* request. Querying of reservations can be performed based on *connectionId* or *globalReservationId*. Filter items specified are OR'ed to build the match criteria. If no criteria are specified then all reservations associated with the RA are returned.



**Figure 58 – *querySummary* request message structure.**

***Parameters***

The *querySummary* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation. Return reservations containing this *connectionId*. |
| *globalReservationId* | An optional global reservation id that can be used to correlate individual related service reservations through the network. Return reservations containing this *globalReservationId*. |

**Table 34 *querySummary*** **message parameters**

***Response***

If the *querySummary* operation is successful, a *querySummaryACK*message is returned, otherwise a *serviceException* is returned. A PA sends this *querySummaryACK*message immediately after receiving the *querySummary* request to acknowledge to the RA the *querySummary* request has been accepted for processing. The *querySummaryACK*message is implemented using the generic acknowledgement message.

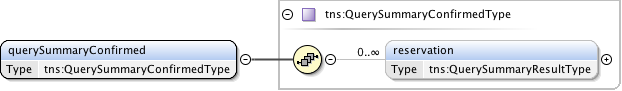


**Figure 59 – *querySummaryACK*** **message structure.**

The *querySummaryACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: querySummaryConfirmed

This *querySummaryConfirmed* message is sent from the PA to RA as an indication of a successful *querySummary* operation. This is in response to an original *querySummary* request from the associated RA.



**Figure 60 – *querySummaryConfirmed* message structure.**

***Parameters***

The *querySummaryConfirmed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *reservation* | A set of zero or more connection reservations matching the query criteria. If there were no matches to the query then no reservation elements will be present. |

**Table 35 *querySummaryConfirmed* message parameters**

***Response***

If the *querySummaryConfirmed* operation is successful, a *querySummaryConfirmedACK*message is returned, otherwise a *serviceException* is returned. A RA sends this *querySummaryConfirmedACK*message immediately after receiving the *querySummaryConfirmed* request to acknowledge to the PA the *querySummaryConfirmed* request has been accepted for processing. The *querySummaryConfirmedACK*message is implemented using the generic acknowledgement message.

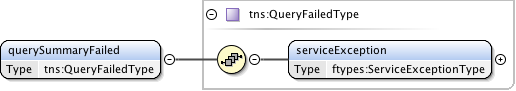


**Figure 61 – *querySummaryConfirmedACK* message structure.**

The *querySummaryConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Failed: querySummaryFailed

This *querySummaryFailed* message is sent from the PA to RA as an indication of a *querySummary* operation failure. This is in response to an original *querySummary* request from the associated RA. It is important to note that a *querySummary* operation that results in no matching reservations does not result in a *querySummaryFailed*message, but instead a*querySummaryConfirmed*with an empty list of reservations.



**Figure 62 – *querySummaryFailed*** **message structure.**

***Parameters***

The *querySummaryFailed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *serviceException* | Specific error condition indicating the reason for the failure. |

**Table 36 *querySummaryFailed* message parameters**

***Response***

If the *querySummaryFailed* operation is successful, a *querySummaryFailedACK* message is returned, otherwise a *serviceException* is returned. A RA sends this *querySummaryFailedACK* message immediately after receiving the *querySummaryFailed* request to acknowledge to the PA the *querySummaryFailed* request has been accepted for processing. The *querySummaryFailedACK* message is implemented using the generic acknowledgement message.



**Figure 63 – *querySummaryFailedACK*** **message structure.**

The *querySummaryFailedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### querySummarySync message elements

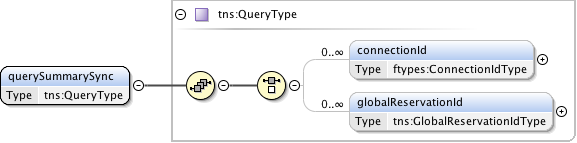
The *querySummarySync* message is sent from a RA to a PA to determine the status of existing reservations on the PA. Unlike the *querySummary* operation, the *querySummarySync* is synchronous and will block until the results of the query operation have been collected. A *querySummarySyncConfirmed* will be returned in response to the request once the query has completed. A *querySummarySyncFailed* message will be sent in response if a processing error has occurred. These responses will be returned directly in the SOAP response to the *querySummarySync* message. Other than the synchronous transport interactions, the *querySummarySync* is identical to the *querySummary* operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *querySummarySync* | *querySummarySyncConfirmed* | *querySummarySyncFailed* |

**Table 37 *querySummarySync*** **message elements**

#### Request: querySummarySync

The *querySummarySync* message provides a mechanism for a RA to query the PA for a set of connection service reservation instances between the RA-PA pair. This message can also be used as a reservation status polling mechanism.  
   
Elements compose a filter for specifying the reservations to return in response to the *querySummarySync* request. Querying of reservations can be performed based on *connectionId* or *globalReservationId*. Filter items specified are OR'ed to build the match criteria. If no criteria are specified then all reservations associated with the RA are returned.



**Figure 64 – *querySummarySync* request message structure.**

***Parameters***

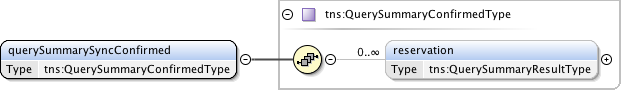
The *querySummarySync* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation. Return reservations containing this *connectionId*. |
| *globalReservationId* | An optional global reservation id that can be used to correlate individual related service reservations through the network. Return reservations containing this *globalReservationId*. |

**Table 38 *querySummarySync*** **message parameters**

***Response (Confirmed)***

If the *querySummarySync* operation is successful, a *querySummarySyncConfirmed* message is returned; otherwise a *querySummarySyncFailed* is returned to indicate an error in processing the query has occurred.



**Figure 65 – *querySummarySyncConfirmed* message structure.**

***Parameters***

The *querySummarySyncConfirmed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *reservation* | A set of zero or more connection reservations matching the query criteria. If there were no matches to the query then no reservation elements will be present. |

**Table 39 *querySummarySyncConfirmed* message parameters**

***Response (Failed)***

This *querySummarySyncFailed* message is sent from the PA to RA as an indication of a *querySummarySync* operation failure. This is in response to an original *querySummarySync* request from the associated RA, and will be returned as a SOAP fault in original request. It is important to note that a *querySummarySync* operation that results in no matching reservations does not result in a *querySummarySyncFailed* message, but instead a *querySummarySyncConfirmed* with an empty list of reservations.



**Figure 66 – *querySummarySyncFailed*** **message structure.**

***Parameters***

The *querySummarySyncFailed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *serviceException* | Specific error condition indicating the reason for the failure. |

**Table 40 *querySummarySyncFailed* message parameters**

### queryRecursive message elements

The *queryRecursive* message is sent from a RA to a PA to determine the status of existing reservations. The *queryRecursiveACK* indicates that the PA has accepted the *queryRecursive* request for processing. A *queryRecursiveConfirmed* or *queryRecursiveFailed* message will be sent asynchronously to the RA when *queryRecursive* processing has completed.

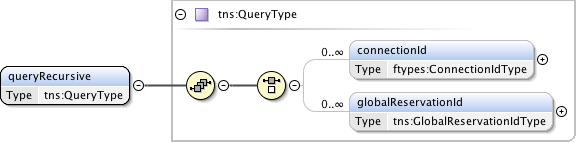
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *queryRecursive* | *queryRecursiveACK* | *serviceException* |
| Confirmed | PA to RA | *queryRecursiveConfirmed* | *queryRecursiveConfirmedACK* | *serviceException* |
| Failed | PA to RA | *queryRecursiveFailed* | *queryRecursiveFailedACK* | *serviceException* |

**Table 41 *queryRecursive*** **message elements**

#### Request: queryRecursive

The *queryRecursive* message provides a mechanism for a RA to query the PA for a set of connection service reservation instances between the RA-PA pair. The returned results will be a detailed list of reservation information collected by recursively traversing the reservation tree.

Elements compose a filter for specifying the reservations to return in response to the *queryRecursive* request. Querying of reservations can be performed based on *connectionId* or *globalReservationId*. Filter items specified are OR'ed to build the match criteria. If no criteria are specified then all reservations associated with the RA are returned.



**Figure 67 – *queryRecursive* request message structure.**

***Parameters***

The *queryRecursive* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | The PA assigned *connectionId* for this reservation. Return reservations containing this *connectionId*. |
| *globalReservationId* | An optional global reservation id that can be used to correlate individual related service reservations through the network. Return reservations containing this *globalReservationId*. |

**Table 42 *queryRecursive*** **message parameters**

***Response***

If the *queryRecursive* operation is successful, a *queryRecursiveACK* message is returned, otherwise a *serviceException* is returned. A PA sends this *queryRecursiveACK* message immediately after receiving the *queryRecursive* request to acknowledge to the RA the *queryRecursive* request has been accepted for processing. The *queryRecursiveACK* message is implemented using the generic acknowledgement message.

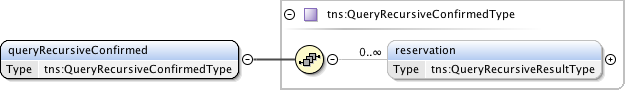


**Figure 68 – *queryRecursiveACK*** **message structure.**

The *queryRecursiveACK* message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: queryRecursiveConfirmed

This *queryRecursiveConfirmed* message is sent from the PA to RA as an indication of a successful *queryRecursive* operation. This is in response to an original *queryRecursive* request from the associated RA.



**Figure 69 – *queryRecursiveConfirmed* message structure.**

***Parameters***

The *queryRecursiveConfirmed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *reservation* | A set of zero or more connection reservations matching the query criteria. If there were no matches to the query then no reservation elements will be present. |

**Table 43 *queryRecursiveConfirmed* message parameters**

***Response***

If the *queryRecursiveConfirmed* operation is successful, a *queryRecursiveConfirmedACK* message is returned, otherwise a *serviceException* is returned. A RA sends this *queryRecursiveConfirmedACK* message immediately after receiving the *queryRecursiveConfirmed* request to acknowledge to the PA the *queryRecursiveConfirmed* request has been accepted for processing. The *queryRecursiveConfirmedACK* message is implemented using the generic acknowledgement message.

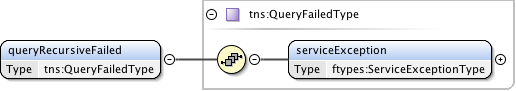


**Figure 70 – *queryRecursiveConfirmedACK* message structure.**

The *queryRecursiveConfirmedACK* message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Failed: queryRecursiveFailed

This *queryRecursiveFailed* message is sent from the PA to RA as an indication of a *queryRecursive* operation failure. This is in response to an original *queryRecursive* request from the associated RA. It is important to note that a *queryRecursive* operation that results in no matching reservations does not result in a *queryRecursiveFailed* message, but instead a *queryRecursiveConfirmed* with an empty list of reservations.



**Figure 71 – *queryRecursiveFailed*** **message structure.**

***Parameters***

The *queryRecursiveFailed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *serviceException* | Specific error condition indicating the reason for the failure. |

**Table 44 *queryRecursiveFailed* message parameters**

***Response***

If the *queryRecursiveFailed* operation is successful, a *queryRecursiveFailedACK* message is returned, otherwise a *serviceException* is returned. A RA sends this *queryRecursiveFailedACK* message immediately after receiving the *queryRecursiveFailed* request to acknowledge to the PA the *queryRecursiveFailed* request has been accepted for processing. The *queryRecursiveFailedACK* message is implemented using the generic acknowledgement message.



**Figure 72 – *queryRecursiveFailedACK*** **message structure.**

The *queryRecursiveFailedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### queryNotification message elements

The *queryNotification* message is sent from a RA to a PA to retrieve notifications messages against an existing reservation residing on the PA. The returned results will be a list of notifications for the specified *connectionId*.

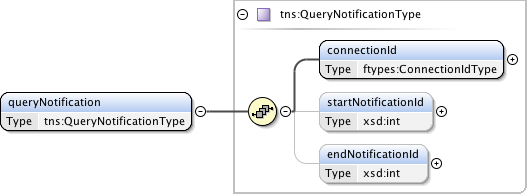
The *queryNotificationACK* indicates that the PA has accepted the *queryNotification* request for processing. A *queryNotificationConfirmed* or *queryNotificationFailed* message will be sent asynchronously to the RA when *queryNotification* processing has completed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| Request | RA to PA | *queryNotification* | *queryNotificationACK* | *serviceException* |
| Confirmed | PA to RA | *queryNotificationConfirmed* | *queryNotificationConfirmedACK* | *serviceException* |
| Failed | PA to RA | *queryNotificationFailed* | *queryNotificationFailedACK* | *serviceException* |

**Table 45 *queryNotification*** **message elements**

#### Request: queryNotification

The *queryNotification* message provides a mechanism for a RA to query the PA for a list of notification messages against a *connectionId*. This operation can be used to recover lost notification messages, or get a historical list of notifications for analysis.  
   
Elements compose a filter for specifying the notifications to return in response to the query operation. The filter query provides an inclusive range of notification identifiers based on *connectionId*.



**Figure 73 – *queryNotification* request message structure.**

***Parameters***

The *queryNotification* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | Notifications for this *connectionId*. |
| *startNotificationId* | The start of the range of *notificationId*s to return. If not present then the query should start from oldest *notificationId* available. |
| *endNotificationId* | The end of the range of *notificationId*s to return. If not present then the query should end with the newest *notificationId* available. |

**Table 46 *queryNotification*** **message parameters**

***Response***

If the *queryNotification* operation is successful, a *queryNotificationACK* message is returned, otherwise a *serviceException* is returned. A PA sends this *queryNotificationACK* message immediately after receiving the *queryNotification* request to acknowledge to the RA the *queryNotification* request has been accepted for processing. The *queryNotificationACK* message is implemented using the generic acknowledgement message.

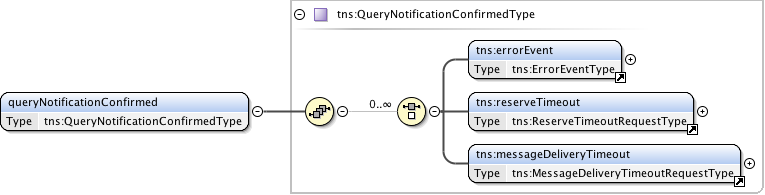


**Figure 74 – *queryNotificationACK*** **message structure.**

The *queryNotificationACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Confirmation: queryNotificationConfirmed

This *queryNotificationConfirmed* message is sent from the PA to RA as an indication of a successful *queryNotification* operation. This is in response to an original *queryNotification* request from the associated RA and contains a list of notification messages matching the query criteria.



**Figure 75 – *queryNotificationConfirmed* message structure.**

***Parameters***

The *queryNotificationConfirmed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *errorEvent* | A set of zero or more error event notifications. |
| *reserveTimeout* | A set of zero or more reserve timeout notification. |
| *messageDeliveryTimeout* | A set of zero or more message delivery timeout notification. |

**Table 47 *queryNotificationConfirmed* message parameters**

***Response***

If the *queryNotificationConfirmed* operation is successful, a *queryNotificationConfirmedACK* message is returned, otherwise a *serviceException* is returned. A RA sends this *queryNotificationConfirmedACK* message immediately after receiving the *queryNotificationConfirmed* request to acknowledge to the PA the *queryNotificationConfirmed* request has been accepted for processing. The *queryNotificationConfirmedACK* message is implemented using the generic acknowledgement message.

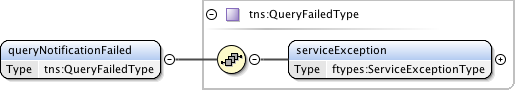


**Figure 76 – *queryNotificationConfirmedACK* message structure.**

The *queryNotificationConfirmedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

#### Failed: queryNotificationFailed

This *queryNotificationFailed* message is sent from the PA to RA as an indication of a *queryNotification* operation failure. This is in response to an original *queryNotification* request from the associated RA. It is important to note that a *queryNotification* operation that results in no matching notifications does not result in a *queryNotificationFailed* message, but instead a *queryNotificationConfirmed* with an empty list.



**Figure 77 – *queryNotificationFailed*** **message structure.**

***Parameters***

The *queryNotificationFailed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *serviceException* | Specific error condition indicating the reason for the failure. |

**Table 48 *queryNotificationFailed* message parameters**

***Response***

If the *queryNotificationFailed* operation is successful, a *queryNotificationFailedACK* message is returned, otherwise a *serviceException* is returned. A RA sends this *queryNotificationFailedACK* message immediately after receiving the *queryNotificationFailed* request to acknowledge to the PA the *queryNotificationFailed* request has been accepted for processing. The *queryNotificationFailedACK* message is implemented using the generic acknowledgement message.



**Figure 78 – *queryNotificationFailedACK*** **message structure.**

The *queryNotificationFailedACK*message has no parameters as all relevant information is carried in the NSI CS header structure.

### queryNotificationSync message elements

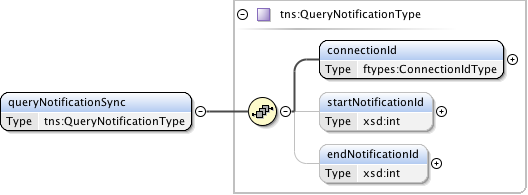
The *queryNotificationSync* message is sent from a RA to a PA to retriever a list of notification messages associated with a *connectionId* on the PA. Unlike the *queryNotification* operation, the *queryNotificationSync* is synchronous and will block until the results of the query operation have been collected. A *queryNotificationSyncConfirmed* will be returned in response to the request once the query has completed. A *queryNotificationSyncFailed* message will be sent in response if a processing error has occurred. These responses will be returned directly in the SOAP response to the *queryNotificationSync* message. Other than the synchronous transport interactions, the *queryNotificationSync* is identical to the *queryNotification* operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Direction | Input | Output | Fault |
| *Request* | RA to PA | *queryNotificationSync* | *queryNotificationSyncConfirmed* | *queryNotificationSyncFailed* |

**Table 49 *queryNotificationSync*** **message elements**

#### Request: queryNotificationSync

The *queryNotificationSync* message provides a mechanism for a RA to query the PA for a list of notification messages against a *connectionId*. This operation can be used to recover lost notification messages, or get a historical list of notifications for analysis.  
   
Elements compose a filter for specifying the notifications to return in response to the query operation. The filter query provides an inclusive range of notification identifiers based on *connectionId*.



**Figure 79 – *queryNotificationSync* request message structure.**

***Parameters***

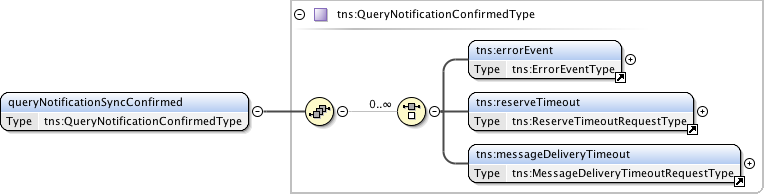
The *queryNotificationSync* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *connectionId* | Notifications for this *connectionId*. |
| *startNotificationId* | The start of the range of *notificationId*s to return. If not present then the query should start from oldest *notificationId* available. |
| *endNotificationId* | The end of the range of *notificationId*s to return. If not present then the query should end with the newest *notificationId* available. |

**Table 50 *queryNotificationSync*** **message parameters**

***Response (Confirmed)***

If the *queryNotificationSync* operation is successful, a *queryNotificationSyncConfirmed* message is returned; otherwise a *queryNotificationSyncFailed* is returned to indicate an error in processing the query has occurred.



**Figure 80 – *queryNotificationSyncConfirmed* message structure.**

***Parameters***

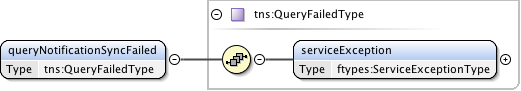
The *queryNotificationSyncConfirmed* message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *errorEvent* | A set of zero or more error event notifications. |
| *reserveTimeout* | A set of zero or more reserve timeout notification. |
| *messageDeliveryTimeout* | A set of zero or more message delivery timeout notification. |

**Table 51 *queryNotificationSyncConfirmed* message parameters**

***Response (Failed)***

This *queryNotificationSyncFailed* message is sent from the PA to RA as an indication of a *queryNotificationSync* operation failure. This is in response to an original *queryNotificationSync* request from the associated RA, and will be returned as a SOAP fault in original request. It is important to note that a *queryNotificationSync* operation that results in no matching notification messages does not result in a *queryNotificationSyncFailed* message, but instead a *queryNotificationSyncConfirmed* with an empty list.



**Figure 81 – *queryNotificationSyncFailed*** **message structure.**

***Parameters***

The *queryNotificationSyncFailed*message has the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| *serviceException* | Specific error condition indicating the reason for the failure. |

**Table 52 *queryNotificationSyncFailed* message parameters**

## NSI CS specific types

***Namespace definition:*** <http://schemas.ogf.org/nsi/2013/04/connection/types>

This section describes the connection services types used for the CS operation definitions.

### Complex Types

These complex type definitions are utilized by the CS operations and are structures containing other elements and/or attributes. Types are listed in alphabetical order.

#### ChildRecursiveListType

A holder element providing an envelope that will contain the list of child NSA and associated detailed connection information. Utilized by the *QueryRecursiveResultType* to provide a nested list structure of detailed path information.



**Figure 82 – *ChildRecursiveListType*.**

***Parameters***

The *ChildRecursiveListType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *child* | O | Detailed path information for a child NSA. Each child element is ordered and contains a connection segment in the overall path. |

**Table 53 *ChildRecursiveListType* message parameters**

#### ChildSummaryListType

A holder element containing a list of children NSA and their associated connection information. Utilized by the *QuerySummaryResultType* to provide a nested list structure of summary path information.



**Figure 83 – *ChildSummaryListType*.**

***Parameters***

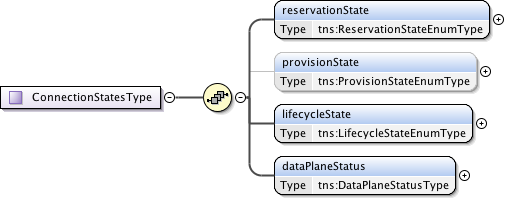
The *ChildSummaryListType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *child* | O | Summary path information for a child NSA. Each child element is ordered and contains a connection segment in the overall path. |

**Table 54 *ChildSummaryListType* message parameters**

#### ConnectionStatesType

A holder element containing the state machines associated with a connection reservation.



**Figure 84 – *ConnectionStatesType*.**

***Parameters***

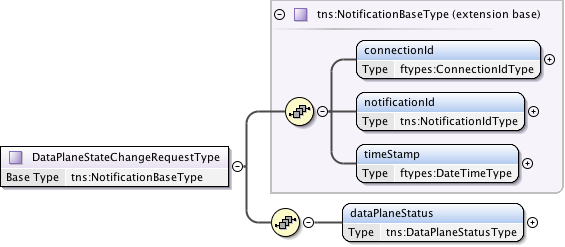
The *ConnectionStatesType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *reservationState* | M | Models the current connection reservation state. |
| *provisionState* | O | Models the current connection provisioning state. The provisionState is created for a reservation once the reservation is committed. |
| *lifecycleState* | M | Models the current connection lifecycle state. |
| *dataPlaneStatus* | M | Models the current connection data plane activation state. |

**Table 55 *ConnectionStatesType* message parameters**

#### DataPlaneStateChangeRequestType

Type definition for the data plane state change notification message.  
   
This notification message sent up from a PA when a data plane status has changed. Possible data plane status changes are: activation, deactivation and activation version change.



**Figure 85 – *DataPlaneStateChangeRequestType*.**

***Parameters***

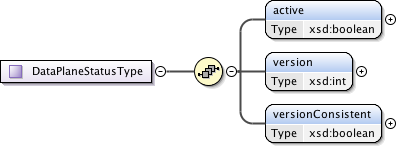
The *DataPlaneStateChangeRequestType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The reservation experiencing the data plane state change. |
| *notificationId* | M | A notification identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for ordering notifications in the context of the *connectionId*. |
| *timeStamp* | M | Time the event was generated on the originating NSA. |
| *dataPlaneStatus* | M | Current data plane activation state for the reservation identified by *connectionId*. |

**Table 56 *DataPlaneStateChangeRequestType* message parameters**

#### DataPlaneStatusType

Models the current connection activation state within the data plane.



**Figure 86 – *DataPlaneStatusType*.**

***Parameters***

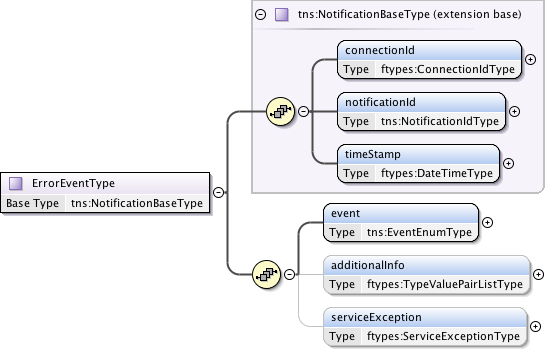
The *DataPlaneStatusType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *active* | M | True if the dataplane is active. For an aggregator, this flag is true when data plane is activated in all participating children. |
| *version* | M | Version of the connection reservation this entry is modeling. |
| *versionConsistent* | M | Always true for uPA. For an aggregator, if version numbers of all children are the same. This flag is true. This field is valid when Active is true. |

**Table 57 *DataPlaneStatusType* message parameters**

#### ErrorEventType

Type definition for an autonomous message issued from a PA to a RA when an existing reservation encounters an autonomous error condition such as being administratively terminated before the reservation's scheduled end-time.



**Figure 87 – *ErrorEventType*.**

***Parameters***

The *ErrorEventType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. |
| *notificationId* | M | A notification identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for ordering notifications in the context of the *connectionId*. |
| *timeStamp* | M | Time the event was generated on the originating NSA. |
| *event* | M | The type of event that generated this notification. |
| *additionalInfo* | O | Type/value pairs that can provide additional error context as needed. |
| *serviceException* | O | Specific error condition - the reason for the generation of the error event. |

**Table 58 *ErrorEventType* message parameters**

#### GenericAcknowledgmentType

A common acknowledgment message type definition. The *correlationId* has been moved to the header in CS version 2 so this is now an empty response.



**Figure 88 – *GenericAcknowledgmentType*.**

Notes on acknowledgment:

Depending on NSA implementation and thread timing an acknowledgment to a request operation may be returned after a confirm/fail for the request has been returned to the RA. For protocol robustness, the RA should be able to accept confirm/fail before acknowledgment.

#### GenericConfirmedType

This is a generic type definition for a Confirmed messages in response to a successful processing of a previous Request message such as *provision*, *release*, and *terminate*.



**Figure 89 – *GenericConfirmedType*.**

***Parameters***

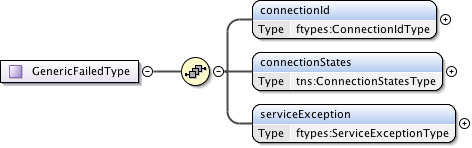
The *GenericConfirmedType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation request. This value will be unique within the context of the PA. |

**Table 59 *GenericConfirmedType* message parameters**

#### GenericFailedType

A generic failed message type sent as request in response to a failure to process a previous protocol request message. This is used in response to all request types that can return an error.



**Figure 90 – *GenericFailedType*.**

***Parameters***

The *GenericFailedType* has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation request. This value will be unique within the context of the PA. |
| *connectionStates* | M | Overall connection state for the reservation. |
| *serviceException* | M | Specific error condition - the reason for the failure. |

**Table 60 *GenericFailedType*** **message parameters**

#### GenericRequestType

This is a generic type definition for request messages such as *provision*, *release*, and *terminate* that only need a *connectionId* as a request parameter.



**Figure 91 – *GenericRequestType*.**

***Parameters***

The *GenericRequestType*has the following parameters (M = Mandatory, O = Optional):

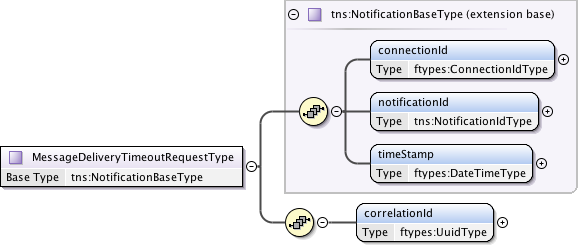
|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation request. This value will be unique within the context of the PA. |

**Table 61 *GenericRequestType* message parameters**

#### MessageDeliveryTimeoutRequestType

A notification message type definition for the Message Transport Layer (MTL) delivery timeout of a request message. In the event of an MTL timed out or Coordinator timeout, the Coordinator will generate this message delivery failure notification and send it up the workflow tree (towards the uRA).

An MTL timeout can be generated as the result of a timeout on receiving an ACK message for a corresponding send request. A Coordinator timeout can occur when no confirm or fail reply has been received to a previous request issued by the Coordinator. In both cases the local timers for these timeout conditions are locally defined.



**Figure 92 – *MessageDeliveryTimeoutRequestType*.**

***Parameters***

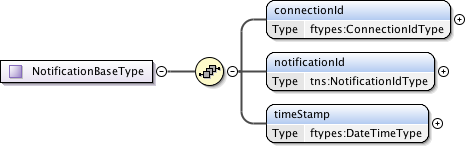
The *MessageDeliveryTimeoutRequestType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The reservation experiencing the data plane state change. |
| *notificationId* | M | A notification identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for ordering notifications in the context of the *connectionId*. |
| *timeStamp* | M | Time the event was generated on the originating NSA. |
| *correlationId* | M | This value indicates the *correlationId* of the original message that the transport layer failed to send. |

**Table 62 *MessageDeliveryTimeoutRequestType* message parameters.**

#### NotificationBaseType

A base type definition for an autonomous message issued from a PA to a RA.



**Figure 93 – *NotificationBaseType*.**

***Parameters***

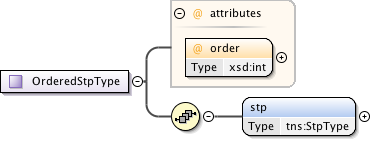
The *NotificationBaseType* has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The reservation experiencing the data plane state change. |
| *notificationId* | M | A notification identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for ordering notifications in the context of the *connectionId*. |
| *timeStamp* | M | Time the event was generated on the originating NSA. |

**Table 63 *NotificationBaseType* message parameters.**

#### OrderedStpType

A Service Termination Point (STP) that can be ordered in a list for use in *PathObject* definition.



**Figure 94 – *OrderedStpType*.**

***Parameters***

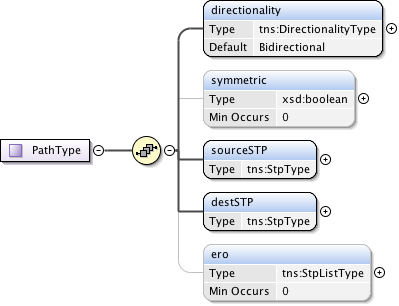
The *OrderedStpType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *order* | M | Order attribute is provided only when the STP is part of an *orderedSTP*List. |
| *stp* | M | The Service Termination Point (STP). |

**Table 64 *OrderedStpType* message parameters**

#### pathType

*Path* of the service is represented by a list of STPs.



**Figure 95 – *PathType*.**

***Parameters***

The *pathType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *directionality* | M | The (uni or bi) directionality of the service. |
| *symmetricPath* | O | An indication that both directions of a bidirectional circuit MUST follow the same path. Only applicable when directionality is *Bidirectional*. If not specified then value is assumed to be false. |
| *sourceSTP* | M | Source STP of the service. |
| *destSTP* | M | Destination STP of the service. |
| *ero* | O | Hop-by-hop ordered list of STPs from *sourceSTP* to *destSTP*. List does not include *sourceSTP* and *destSTP*. |

**Table 65 *PathType* message parameters**

#### QueryFailedType

A query failed message type sent as request in response to a failure to process a *queryRequest* message. This is message is returned as a result of a processing error and not for the case where a query returns an empty result set.



**Figure 96 – *QueryFailedType*.**

***Parameters***

The *QueryFailedType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *ServiceException* | M | Specific error condition - the reason for the failure. |

**Table 66 *pathType* message parameters**

#### QueryRecursiveConfirmedType

This is the type definition for the *queryRecursiveConfirmed* message. An NSA sends this positive *queryRecursive*Request response to the NSA that issued the original request message. There can be zero or more results retuned in this confirmed message depending on the query parameters supplied in the request.



**Figure 97 – *QueryRecursiveConfirmed*Type.**

***Parameters***

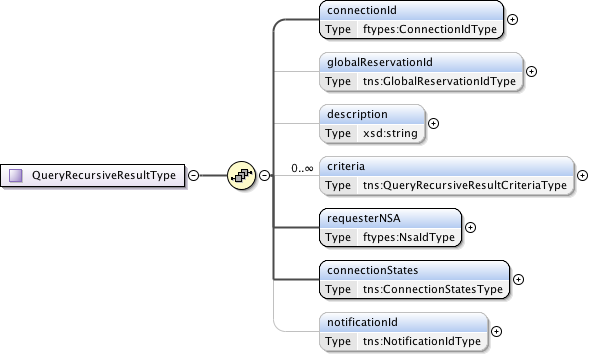
The *QueryRecursiveConfirmed*Typehas the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *reservation* | O | Resulting recursive set of connection reservations matching the query criteria. If there were no matches to the query then no reservation elements will be present. |

**Table 67 *QueryRecursiveConfirmed*Type message parameters**

#### QueryRecursiveResultType

This type contains the common reservation elements and detailed path data for recursive query results.



**Figure 98 – *QueryRecursiveResultType*.**

***Parameters***

The *QueryRecursiveResultType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. |
| *globalReservationId* | O | An optional global reservation id that can be used to correlate individual related service reservations through the network. This MUST be populated with a Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 | ISO/IEC 9834-8:2005 and IETF RFC 4122. |
| *description* | O | An optional description for the service reservation. |
| *criteria* | O | A set of versioned reservation criteria information. |
| *requesterNSA* | M | The RA associated with the reservation. |
| *connectionStates* | M | The reservation's overall connection states. |
| *notificationId* | O | A notification identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for ordering notifications in the context of the *connectionId*. This parameter is present when there is an error notification against this reservation. |

**Table 68 *QueryRecursiveResultType* message parameters.**

#### QuerySummaryConfirmedType

This is the type definition for the *querySummaryConfirmed* message (both synchronous and asynchronous versions). An NSA sends this positive *querySummary*Request response to the NSA that issued the original request message. There can be zero or more results retuned in this confirmed message depending on the number of matching reservation results.



**Figure 99 – *QuerySummaryConfirmed*Type.**

***Parameters***

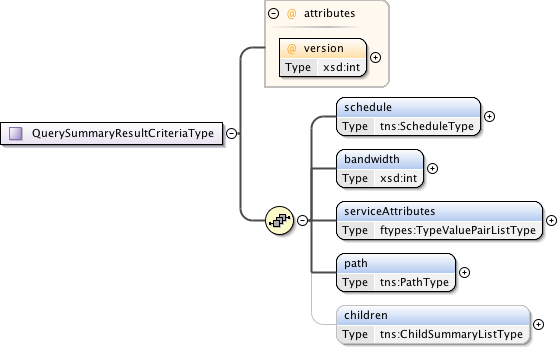
The *QuerySummaryConfirmed*Typehas the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *reservation* | O | Resulting recursive set of connection reservations matching the query criteria. If there were no matches to the query then no reservation elements will be present. |

**Table 69 *QuerySummaryConfirmed*Type message parameters.**

#### QuerySummaryResultCriteriaType

Type definition for the query summary result containing versioned reservation information and associated child connection identifiers.



**Figure 100 – *QuerySummaryResultCriteriaType*.**

***Parameters***

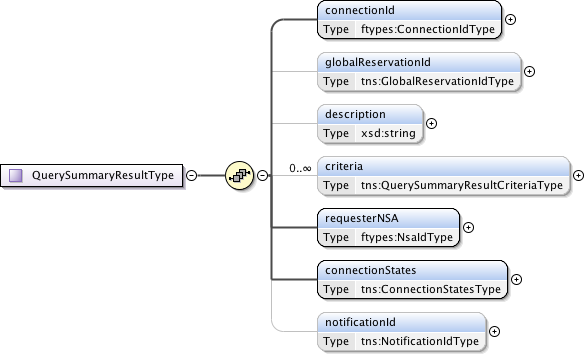
The *QuerySummaryResultCriteriaType* has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *version* | M | Version of the reservation instance. |
| *schedule* | M | Time parameters specifying the life of the service. |
| *bandwidth* | M | Bandwidth of the service in Mb/s. |
| *serviceAttributes* | M | Technology specific attributes relating to the service. |
| *path* | M | The source and destination end points of the service. Can optionally provide additional path segments to guide path computation. |
| *children* | O | If this Connection reservation is aggregating child connections then this element contains basic summary information about the child Connection segment. |

**Table 70 *QuerySummaryConfirmed*Type message parameters.**

#### QuerySummaryResultType

Type containing the set of reservation parameters associated with a summary query result.



**Figure 101 – *QuerySummaryResultType*.**

***Parameters***

The *QuerySummaryResultType*has the following parameters (M = Mandatory, O = Optional):

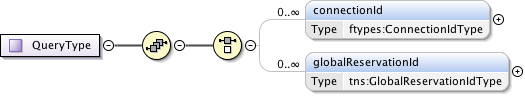
|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. |
| *globalReservationId* | O | An optional global reservation id that can be used to correlate individual related service reservations through the network. This MUST be populated with a Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 | ISO/IEC 9834-8:2005 and IETF RFC 4122. |
| *description* | O | An optional description for the service reservation. |
| *criteria* | O | A set of versioned reservation criteria information. |
| *requesterNSA* | M | The RA associated with the reservation. |
| *connectionStates* | M | The reservation's overall connection states. |
| *notificationId* | M | A notification identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for ordering notifications in the context of the *connectionId*. |

**Table 71 *QuerySummaryResultType* message parameters**

#### QueryType

Type definition for the *querySummary* message providing a mechanism for either RA or PA to query the other NSA for a set of Connection service reservation instances between the RA-PA pair. This message can also be used as a status polling mechanism.

Elements compose a filter for specifying the reservations to return in response to the *queryRequest*. Supports the querying of reservations based on *connectionId* or *globalReservationId*. Filter items specified are OR'ed to build the match criteria. If no criteria are specified then all reservations associated with the RA are returned.



**Figure 102 – *QueryType*.**

***Parameters***

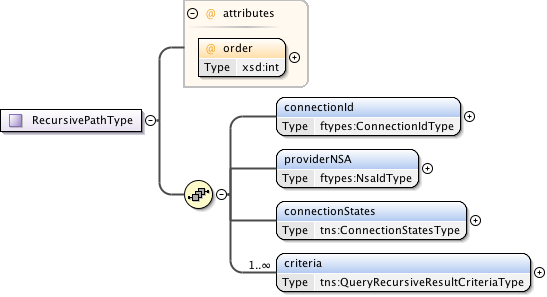
The *QueryType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | O | Return reservations containing this *connectionId*. |
| *globalReservationId* | O | Return reservations containing this *globalReservationId*. |

**Table 72 *QueryType* message parameters**

#### RecursivePathType

This type is used to model a connection reservation's detailed  path information. The structure is recursive so it is possible to model both an ordered list of connection segments, as well as the hierarchical connection segments created on children NSA in either a tree and chain configuration.



**Figure 103 – *RecursivePathType*.**

***Parameters***

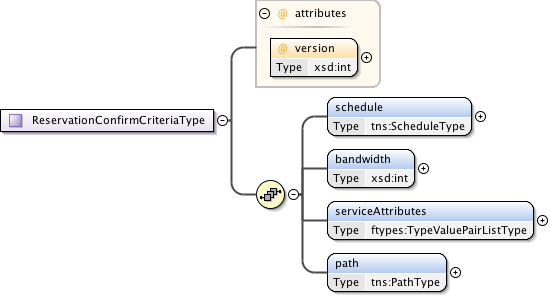
The *RecursivePathType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *order* | M | Specification of ordered path elements. |
| *connectionId* | M | The connection identifier associated with the reservation and path segment. |
| *providerNSA* | M | The PA holding the connection information associated with this instance of data. |
| *connectionStates* | M | This reservation's segments connection states. |
| *criteria* | M | A set of versioned reservation criteria information. |

**Table 73 *RecursivePathType* message parameters**

#### ReservationConfirmCriteriaType

A type definition for the reservation confirmation information used by PA to return reservation information to an RA. Includes the reservation version id to track version of the reservation criteria.



**Figure 104 – *ReservationConfirmCriteriaType*.**

***Parameters***

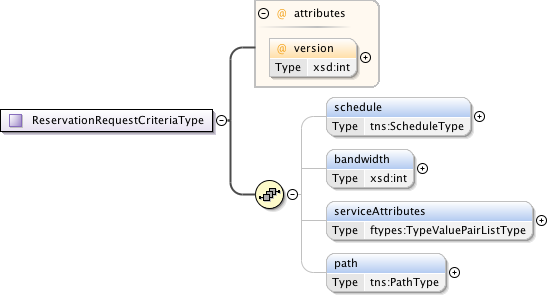
The *ReservationConfirmCriteriaType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *version* | M | Version of the reservation instance. |
| *schedule* | M | Time parameters specifying the life of the service. |
| *bandwidth* | M | Bandwidth of the service in Mb/s. |
| *serviceAttributes* | M | Technology specific attributes relating to the service. |
| *path* | M | The source and destination end points of the service. Can optionally provide additional path segments to guide path computation. |

**Table 74 *ReservationConfirmCriteriaType* message parameters**

#### ReservationRequestCriteriaType

Type definition for a reservation and modification request criteria. Only those values requiring change are specified in the modify request. The *version* value specified in a reservation or modify request MUST be a positive integer larger than the previous *version* number. A *version* value of zero is a special number indicating an allocated but not yet reserved reservation and cannot be specified by the RA.



**Figure 105 – *ReservationRequestCriteriaType*.**

***Parameters***

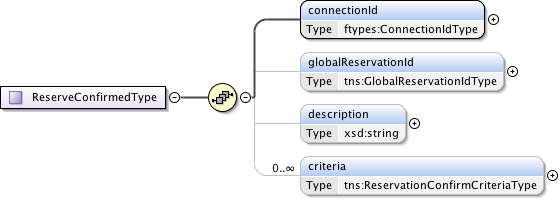
The *ReservationRequestCriteriaType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *version* | M | The version number assigned by the RA to this reservation instance. If not specified in the initial reservation request, the new reservation will default to one for the first version; however, an initial request can specify any positive integer except zero. Each further reservation request on an existing reservation (a modify operation), will be assigned a linear increasing number, either specified by the RA, or assigned by the PA if not specified. |
| *schedule* | M | Time parameters specifying the life of the service. |
| *bandwidth* | M | Bandwidth of the service in Mb/s. |
| *serviceAttributes* | M | Technology specific attributes relating to the service. |
| *path* | M | The source and destination end points of the service. Can optionally provide additional path segments to guide path computation. |

**Table 75 *ReservationRequestCriteriaType* message parameters**

#### ReserveConfirmedType

Type definition for the *reserveConfirmed* message. A PA sends this positive *reserve* request response to the RA that issued the original request message.



**Figure 106 – *ReserveConfirmedType*.**

***Parameters***

The *ReserveConfirmedType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. |
| *globalReservationId* | O | An optional global reservation id that can be used to correlate individual related service reservations through the network. This MUST be populated with a Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 | ISO/IEC 9834-8:2005 and IETF RFC 4122. |
| *description* | O | An optional description for the service reservation. |
| *criteria* | O | A set of versioned reservation criteria information. |

**Table 76 *ReserveConfirmedType* message parameters**

#### ReserveResponseType

Type definition for the *reserveResponse* message. A PA sends this *reserveResponse* message immediately after receiving the *reserve* request to inform the RA of the *connectionId* allocated to their *reserve* request. This *connectionId* can then be used to query reservation progress.



**Figure 107 – *ReserveResponseType*.**

***Parameters***

The ***ReserveResponseType*** has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. |

**Table 77 *ReserveResponseType* message parameters**

#### ReserveTimeoutRequestType

This is the type definition for the reserve timeout notification message. This is an autonomous message issued from a PA to a RA when a timeout on an existing *reserve* request occurs and uncommitted resources have been freed. The type of event originates from a uPA, and is propagated up the request tree to the uRA. The aggregator NSA will map the received *connectionId* into a context understood by the next parent NSA in the request tree, then propagate the event upwards. The originating *connectionId* and NSA are provided in separate elements to maintain the original context generating the timeout. The *timeoutValue* and *timeStamp* are populated by the originating NSA and propagated up the tree untouched.



**Figure 108 – *ReserveTimeoutRequestType*.**

***Parameters***

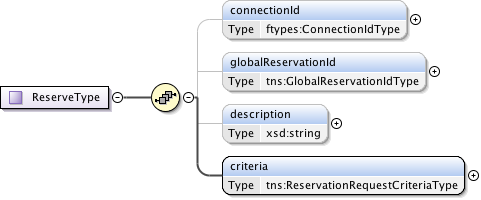
The *ReserveTimeoutRequestType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | M | The reservation experiencing the data plane state change. |
| *notificationId* | M | A notification identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for ordering notifications in the context of the *connectionId*. |
| *timeStamp* | M | Time the event was generated on the originating NSA. |
| *timeoutValue* | M | The timeout value in seconds that expired this reservation. |
| *originatingConnectionId* | M | The *connectionId* that triggered the reserve timeout. |
| *originatingNSA* | M | The NSA originating the timeout event. |

**Table 78 *ReserveTimeoutRequestType* message parameters**

#### ReserveType

This is the type definition that models the reserve message that allows a RA to reserve network resources for a Connection between two STP's constrained by a certain service parameters. This operation allows an RA to check the feasibility of Connection reservation or a modification to an existing reservation. Any resources associated with the reservation or modification will be allocated and held until commit is received or timeout occurs.



**Figure 109 – *ReserveType*.**

***Parameters***

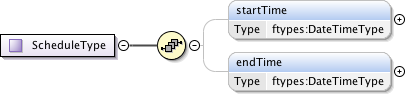
The *ReserveType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *connectionId* | O | The PA assigned *connectionId* for this reservation. This value will be unique within the context of the PA. Provided in reserve request only when an existing reservation is being modified. |
| *globalReservationId* | O | An optional global reservation id that can be used to correlate individual related service reservations through the network. This MUST be populated with a Universally Unique Identifier (UUID) URN as per ITU-T Rec. X.667 | ISO/IEC 9834-8:2005 and IETF RFC 4122. |
| *description* | O | An optional description for the service reservation. |
| *criteria* | M | Reservation request criteria including start and end time, service attributes, and requested path for the service. |

**Table 79 *ReserveType* message parameters**

#### ScheduleType

This type definition models the reservation schedule start and end time parameters.



**Figure 110 – *ScheduleType*.**

***Parameters***

The *ScheduleType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *startTime* | O | Reservation start time. If not specified then immediate reservation. |
| *endTime* | O | Reservation end time. If endTime is not specified then the schedule end is indefinite. |

**Table 80 *ScheduleType* message parameters**

#### StpListType

This type is a simple ordered list type of Service Termination Points (STPs). The list order is determined by the integer order attribute in the *orderedSTP* element.



**Figure 111 – *StpListType*.**

***Parameters***

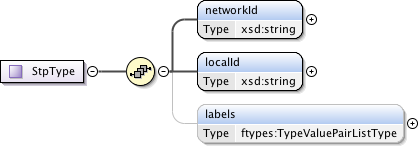
The *StpListType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *orderedSTP* | O | A list of STP ordered 0..n by their integer order attribute. |

**Table 81 *StpListType* message parameters**

#### StpType

This is the Service Termination Point (STP) type used for path selection.



**Figure 112 – *StpType*.**

***Parameters***

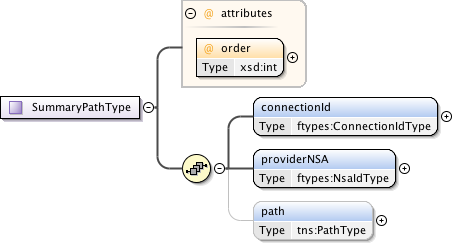
The *StpType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *networkId* | M | A globally unique identifier (URN) that identifies the Network. Rather than forcing parsing of an STP to determine the Network, a separate Network object is defined to allow an intermediate NSA to forward the message to the target Network without needing to know about the STPs within that domain. |
| *localId* | M | A locally unique identifier for the STP within the associated network. |
| *labels* | O | Technology specific attributes associated with the Service Termination Point. |

**Table 82 *StpType* message parameters**

#### SummaryPathType

This type is used to model a connection reservation's summary path information. The structure provides the next level of connection information but not state.



**Figure 113 – *SummaryPathType*.**

***Parameters***

The *SummaryPathType*has the following parameters (M = Mandatory, O = Optional):

|  |  |  |
| --- | --- | --- |
| Parameter | M/O | Description |
| *order* | M | Specification of ordered path elements. |
| *connectionId* | M | The connection identifier associated with the reservation and path segment. |
| *providerNSA* | M | The PA holding the connection information associated with this instance of data. |
| *path* | O | The summary path information associated with the connection reservation. |

**Table 83 *SummaryPathType* message parameters**

### Simple Types

These simple type definitions are utilized by the CS complex type definitions. Types are listed in alphabetical order.

#### DirectionalityType

This type is used to indicate the directionality of the requested data service. Possible values are *Bidirectional* for a bidirectional data service, and *Unidirectional* for a unidirectional data service.



**Figure 114 – *DirectionalityType*.**

#### EventEnumType

Notification event message types. Possible values are:

* *activateFailed* – Indicates that the data plane activation related to a reservation has failed, and therefore, there is no data plane connectivity for the reporting uPA.
* *deactivateFailed* – Indicates that deactivation of the data plane has failed, and as a result, data plane connectivity may still be in place.
* *dataplaneError* – Indicates that an error has occurred in the data plane and a loss of connectivity may be the result.
* *forcedEnd* – Indicates that the reservation was administratively terminated by a PA within the network.



**Figure 115 – *EventEnumType*.**

#### GlobalReservationIdType

A *globalReservationId* is a type representing a globally unique identifier for a reservation. This will be populated with a OGF URN (reference artifact 6478 "Procedure for Registration of Subnamespace Identifiers in the URN:OGF Hierarchy") to be used for compatibility with other external systems.



**Figure 116 – *GlobalReservationIdType*.**

#### LifecycleStateEnumType

Connection lifecycle state values for the reservation lifecycle state machine. The lifecycle state machine is instantiated when a reservation is committed. Possible state values are:

* *Created* – A steady state for the lifecycle state machine and the initial state after a reservation has been committed.
* *Failed* – A steady state for the lifecycle state machine that is reached if a forcedEnd error is received from a uPA.
* *Terminating* - A transient state modeling the act of terminating the reservation.
* *Terminated* - A steady state for the lifecycle state machine that is reached when the reservation is terminated by the uRA.



**Figure 117 – *LifecycleStateEnumType*.**

#### NotificationIdType

A specific type for a *notificationId* that is unique in the context of a *connectionId*.



**Figure 118 – *LifecycleStateEnumType*.**

#### ProvisionStateEnumType

Connection provisioning state values for modeling the connection services provision state machine.

The Provision State Machine (PSM) is a simple state machine that transits between the Provisioned and the Released state. An instance of the PSM is created when an initial reservation is committed, and at that time it remains in the Released state. The PSM transits states independent of the state of the Reservation State Machine. Note that staying at the Provisioned state is necessary but not sufficient to activate the data plane. The data plane is active if the PSM is in Provisioned state AND current\_time is between startTime and endTime.

Possible state values are:

* *Released* – A steady state for the provision state machine in which data plane resources for this reservation are in a released state, resulting in an inactive data plane.
* *Provisioning* - A transient state modeling the act of provisioning the reservation’s associated data plane resources.
* *Provisioned* - A steady state for the provision state machine in which data plane resources for this reservation are in a provisioned state. This state does not imply that data plane resources are active, but it does indicate that a uPA can active the data plane resources if current\_time is between startTime and endTime.
* *Releasing* - A transient state modeling the act of releasing the reservation’s associated data plane resources.



**Figure 119 – *ProvisionStateEnumType*.**

#### ReservationStateEnumType

Connection reservation state values for the connection services reservation state machine. Possible state values are:

* *ReserveStart* – A steady state for the reservation state machine in which a reservation is created and committed. In the case of the first reservation request this state represents the initial reservation shell has been committed to database.
* *ReserveChecking* – A transient state modeling the act of checking the feasibility of a new reservation request, or a request to modify an existing reservation.
* *ReserveFailed* – A steady state for the reservation state machine in which the initial reservation or a subsequent modification request has failed.
* *ReserveAborting* - A transient state modeling the act of aborting a pending reservation modify request.
* *ReserveHeld* - A steady state for the reservation state machine in which the initial reservation or a subsequent modification request has successfully had the request resources reserved, but has not yet been committed.
* *ReserveCommitting* - A transient state modeling the act of committing a held set of reservation resources.
* *ReserveTimeout* - A steady state for the reservation state machine in which the held resources have been locally timed out on a uPA, resulting in a transition from the *ReserveHeld* to *ReserveTimeout* state.



**Figure 120 – ReservationStateEnumType.**

# Security

This section describes how NSI CS protocol achieves secure communication and provides authentication data across requests. Security is achieved using TLS between NSAs and SAML attributes to convey information regarding request authentication.

## Transport Layer Security

TLS is used to ensure secure communication between NSAs and X.509 certificates for authentication. Trust between NSAs is pairwise and MUST be established out-of-band. It is possible to have unidirectional trust between NSAs, i.e. reservations can only be created in one direction, as this is simply a policy special case. Transitive trust between NSAs cannot be assumed, i.e., NSAs A & B trust each other, and B & C trust each other, but this does not imply trust between A & C. However a request from A may end up using resources from C if passed through B. In the current security framework, B (if its policies permit) can proxy A’s request to C. From C’s point of view, it receives the request from B, and authenticates and authorizes the request using B’s credentials. This document does not describe security policies, as these will always be site specific. Note that due to the requirement for direct NSA to NSA communications (i.e. NSAs cannot forward communications via a third party NSA), message-level signing provides little value and is not used.

TLS provides message integrity, confidentiality, protects against replay attacks, and authentication via the X.509 certificates. Authorization is done at the NSAs application level. TLS version 1.0 MUST be supported. NSAs MAY use SSLv3 and TLS version following 1.0 were possible.

## SAML Assertions

As TLS by design only provides transport level security, an additional mechanism for conveying request authentication is required. For this, SAML assertions are used. NSAs can include SAML assertions in the CS message header, which providers MAY use to authorize the request. SAML attributes can describe information such as user, group, originating NSA, or even OAuth tokens. What and how to describe with SAML headers is outside the scope of this document, but will be described in a best current practices (BCP) document. The intent of such a document is to provide a baseline of what to support, but attributes can be created as needed and can be unique to NSA peerings.

# Contributors

Chin Guok, ESnet

Jeroen van der Ham, University of Amsterdam

Radek Krzywania, PSNC

Tomohiro Kudoh, AIST

John MacAuley, SURFnet

Takahiro Miyamoto, KDDI R&D Laboratories

Inder Monga, ESnet

Guy Roberts, DANTE

Jerry Sobieski, NORDUnet

Henrik Thostrup Jensen, NORDUnet

# Glossary

|  |  |
| --- | --- |
| Activate | When provisioning of a Connection has been completed the Connection is considered to be Active. A notification is sent to the RA informing them that the Connection is Active. |
| Aggregator (AG) | The Aggregator is an NSA that has more than one child NSA, and has the responsibility of aggregating the responses from each child NSA. |
| Connection | A Connection is an NSI construct that identifies the physical instance of a circuit in the data plane. A Connection has a set of properties (for instance, Connection identifier, ingress and egress STPs, capacity, or start time). Connections can be either unidirectional or bidirectional. |
| Connection Service (CS) | The NSI Connection Service is a service that allows a RA to request and manage a Connection from a PA. |
| Connection Service Protocol | The Connection Service Protocol is the protocol that describes the messages and associated attributes that are exchanged between RA and PA. |
| Control and Management Planes | The Control Plane and/or Management Plane are not defined in this document, but follow common usage. |
| Coordinator | The Coordinator function has the role of providing intelligent message and process coordination, this includes tracking and aggregating messages, replies and notifications and the servicing of query requests. |
| Data Plane | The Data Plane refers to the infrastructure that carries the physical instance of the Connection, e.g. the Ethernet switches that deliver the circuit. |
| Discovery Service | The NSI discovery service is a web service that allows an RA to discover information about the services available in a PA and the versions of these services. |
| Edge Point | A network resource that resides at the boundary of an intra-network topology, this may include for example a connector on a distribution frame, a port on an Ethernet switch, or a connector at the end of a fibre. |
| Inter-Network Topology | This is a topological description of a set of Networks and their transfer functions, and the connectivity between Networks |
| Lifecycle State Machine (LSM) | The LSM allows messages relating to terminating a Connection to be to send and received. |
| Message Transport Layer (MTL) | The MTL delivers an abstracted message delivery mechanism to the NSI layer. |
| Network | A Network is an Inter-Network topology object that describes a set of STPs with a Transfer Function between STPs |
| Network Resource Manager (NRM) | The Network Resource Manager owns a set of transport resources and has ultimate responsibility for authorizing and managing the use of these resources. Each NRM is always associated with a single NSA |
| Network Services | Network Services are the full set of services offered by an NSA. Each NSA will support one or more Network Services |
| Network Service Agent (NSA) | The Network Service Agent is a concrete piece of software that sends and receives NSI Messages. The NSA includes a set of capabilities that allow Network Services to be delivered. |
| Network Service Interface (NSI) | The NSI is the interface between RAs and PAs. The NSI defines a set of interactions or transactions between these NSAs to realize a Network Service |
| Network Services Framework (NSF) | The Network Services framework describes a NSI message based platform capable of supporting a suite of Network Services such as the Connection Service and the Topology Service |
| NSI Message | A NSI Message is a structured unit of data sent between a RA and a PA. |
| Path | A Path is an ordered list of STPs that are used a constraint in Inter-Network pathfinding. |
| Provision | Provisioning is the process of requesting the creation of the physical instance of a Connection in the data plane |
| Provision State Machine (PSM) | The Provision State Machine is a simple state machine which transits between the Provisioned and the Released state |
| Release | Releasing is the process of de-provisioning resources on the data-plane. When a Connection is Released on the data-plane, the Reservation is retained. |
| Requester/ PA (RA/PA) | An NSA acts in one of two possible roles relative to a particular instance of an NSI. When an NSA requests a service, it is called a Requester Agent (RA). When an NSA realizes a service, it is called a Provider Agent (PA). A particular NSA may act in different roles at different interfaces |
| Reservation State Machine (RSM) | The state machine that defines the message sequence for creating Connection reservations and managing these reservations. |
| Service Demarcation Point (SDP) | Service Demarcation Points (SDPs) are an NSI topology objects that identify a grouping of two Edge Points at the boundary between two Networks. |
| Service Termination Point (STP) | Service Termination Points (STPs) are an NSI topology objects that identify the Edge Points of a Network in the intra-network topology. |
| Service Plane | The Service Plane is a plane in which services are requested and managed; these services include the Network Service. The Service Plane contains a set of Network Service Agents communicating using Network Service Interfaces |
| Simple Object Access Protocol (SOAP) | SOAP is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks |
| Reservation State Machine (RSM) | The Reservation State Machine state machine defines the sequence of operation of messages for creating or modifying a reservation |
| Reserve | When a Provider Agent receives (and then confirms) a Connection Reservation request the Provider Agent then holds the resources needed by the Connection. |
| Topology Distribution Service | The NSI Topology distribution Service is a service that allows the NSI topology to be exchanged between NSAs |
| Terminate | Terminating is the process which will completely remove a Reservation and Release any associated Connections. This term has a formal definition in the CS state-machine |
| Ultimate PA (uPA) | The ultimate PA is a Provider Agent that has an associated NRM. |
| Ultimate RA (uRA) | The Ultimate RA is a Requester Agent is the originator of a service request |
| XML Schema Definition (XSD) | XSD is a schema language for XML |
| eXtensible Markup Language (XML) | XML is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. |

# Intellectual Property Statement

The OGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat.

The OGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the OGF Executive Director.

# Disclaimer

This document and the information contained herein is provided on an “As Is” basis and the OGF disclaims all warranties, express or implied, including but not limited to any warranty that the use of the information herein will not infringe any rights or any implied warranties of merchantability or fitness for a particular purpose.

# Full Copyright Notice

Copyright (C) Open Grid Forum (2008-2013). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the OGF or other organizations, except as needed for the purpose of developing Grid Recommendations in which case the procedures for copyrights defined in the OGF Document process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the OGF or its successors or assignees.

# Appendix A: State Machine Transition Tables

This appendix describes the transitions that are allowed in the NSI CS state machines. These tables should be read in conjunction with the state machines described in section 4.2.



Table 66. RSM transition table



Table 67. PSM transition table



Table 68. LSM transition table

# Appendix B: Error Messages and Best Practices

## Error Messages

The following set of error codes apply.

|  |  |  |  |
| --- | --- | --- | --- |
| ***errorId*** | ***errorDescription*** | ***text*** | ***variables*** |
| 100 | PAYLOAD\_ERROR |  |  |
| 101 | MISSING\_PARAMETER | Invalid or missing parameter | Include the parameter name that is missing. |
| 102 | UNSUPPORTED\_PARAMETER | Parameter provided contains an unsupported value which MUST be processed. | Include the parameter name that is unsupported. |
| 103 | NOT\_IMPLEMENTED |  | Include the capability that is not implemented. |
| 104 | VERSION\_NOT\_SUPPORTED | The service version requested in NSI header is not supported. | Return type *protocolVersion* and value the version requested. |
| 200 | CONNECTION\_ERROR |  |  |
| 201 | INVALID\_TRANSITION | Connection state machine is in invalid state for received message. | Include the current state of the state machine. |
| 202 | CONNECTION\_EXISTS | Schedule already exists for *connectionId* |  |
| 203 | CONNECTION\_NONEXISTENT | Schedule does not exist for *connectionId*. |  |
| 204 | CONNECTION\_GONE |  |  |
| 205 | CONNECTION\_CREATE\_ERROR | Failed to create connection (payload was ok, something went wrong) |  |
| 300 | SECURITY\_ERROR |  |  |
| 301 | AUTHENTICATION\_FAILURE |  |  |
| 302 | UNAUTHORIZED |  |  |
| 400 | TOPOLOGY\_ERROR |  |  |
| 401 | UNKNOWN\_STP | Could not find STP in topology database. | Include the unknown STP. |
| 402 | STP\_RESOLUTION\_ERROR | Could not resolve STP to a managing NSA. | Include the STP that could not be resolved. |
| 403 | NO\_PATH\_FOUND | Path computation failed to resolve route for reservation. |  |
| 404 | VLANID\_INTERCANGE\_NOT\_SUPPORTED | VlanId interchange not supported for requested path. |  |
| 500 | INTERNAL\_ERROR | An internal error has caused a message processing failure. |  |
| 501 | INTERNAL\_NRM\_ERROR | An internal NRM error has caused a message processing failure. | Include information describing the specific NRM error. |
| 600 | RESOURCE\_UNAVAILABLE |  |  |
| 601 | STP\_UNAVALABLE | Specified STP already in use. |  |
| 602 | BANDWIDTH\_UNAVAILABLE | Insufficient bandwidth available for reservation. |  |

Table69: error messages

## NTP servers

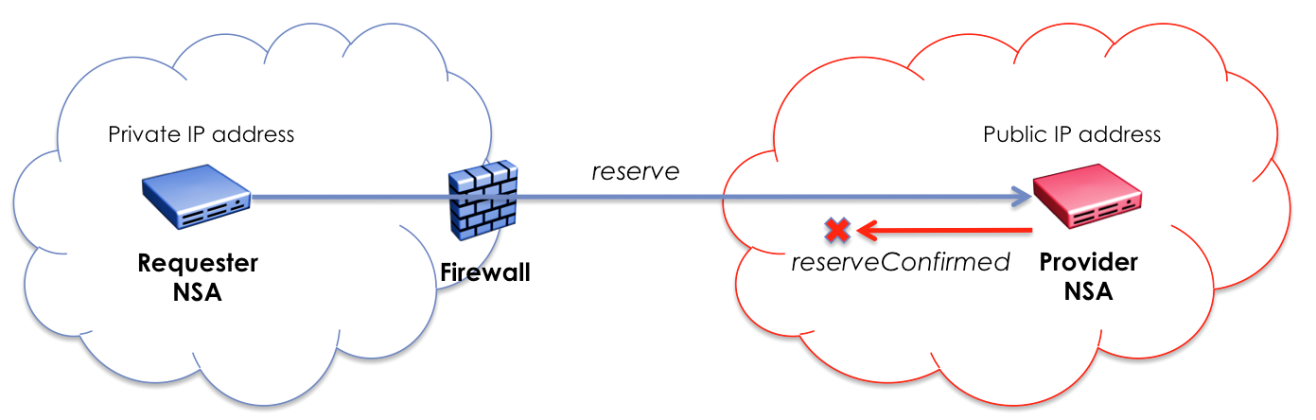
The server running the NSA should use NTP version 4 [8]. This will reduce the risk of clock skew between the NSAs.

# Appendix B: Firewall Handling

Firewalls are commonly disruptive of application level protocols (such as FTP), with specific protocol solutions such as uPnP defined to help applications properly traverse a firewall. The NSI CS HTTP/SOAP binding has similar firewall issues. It is important to maintain appropriate firewall and application configurations for the NSI protocol to function correctly. However, it is recognized there will be situations where an NSA administrator may not be able to influence firewall configurations, and therefore, need an alternative solution.

**Figure 121** shows an example of the common firewall issue that is encountered when deploying an NSA behind a firewall within a private address space. This flow proceeds as follows:

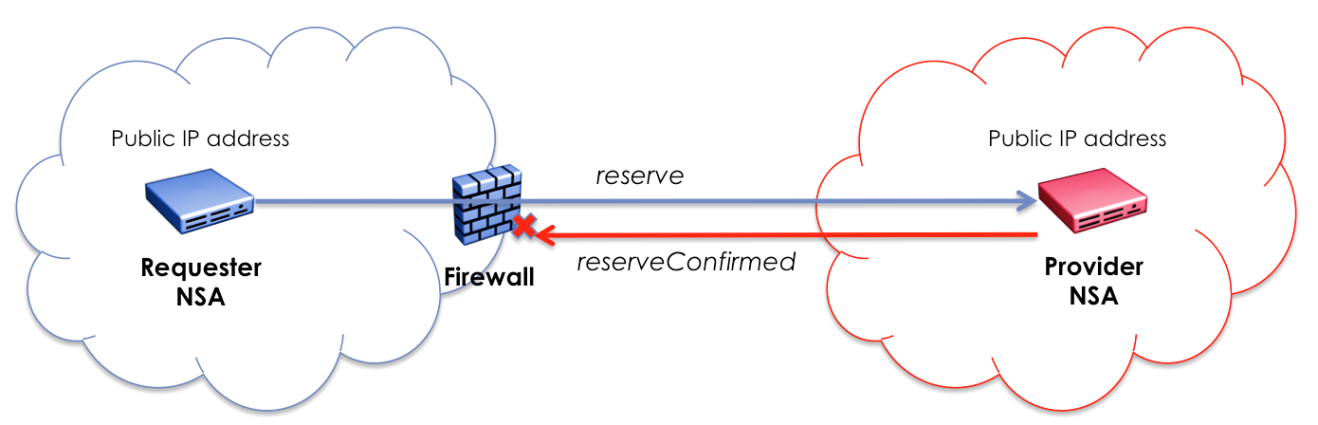
* The RA composes an NSI *reserve* request message populating the *replyTo* field with its SOAP endpoint using private IP address for asynchronous response.
* The RA behind the firewall issues HTTP *reserve* request to PA on the public network.
* The firewall NATs the HTTP request and passes on to the PA but does not NAT the private IP address in *replyTo* since this is embedded in the SOAP message.
* The PA is unable to reach the private IP address to deliver the *reserveConfirmed* message.



**Figure 121 –** RA **(RA) behind a firewall with private IP address.**

Similar issues can occur when the RA is assigned a public IP address but is behind a firewall not configured to forward HTTP traffic to the callback endpoint. **Figure 122** shows an example of this specific issue. This flow proceeds as follows:

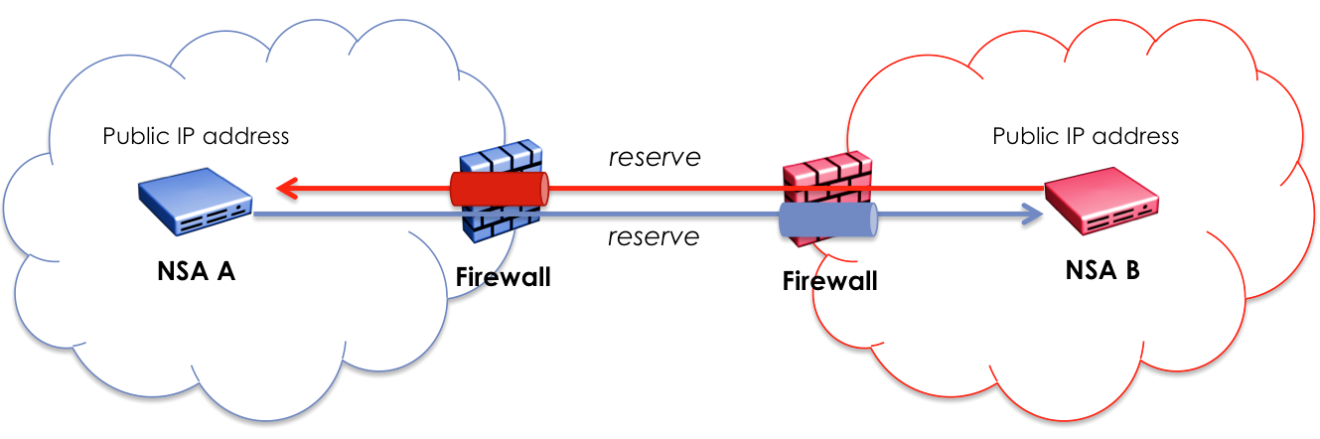
* The RA composes an NSI *reserve* request message populating the *replyTo* field with its SOAP endpoint using public IP address for asynchronous response.
* The RA behind the firewall issues the HTTP *reserve* request to the PA on the public network.
* The firewall passes the request on to PA but requires no NATing of addresses.
* The PA cannot reach the public IP address of the RA to deliver the *reserveConfirmed* message as the firewall is blocking incoming HTTP connections.



**Figure 122 –** RA **(RA) behind a firewall with public IP address.**

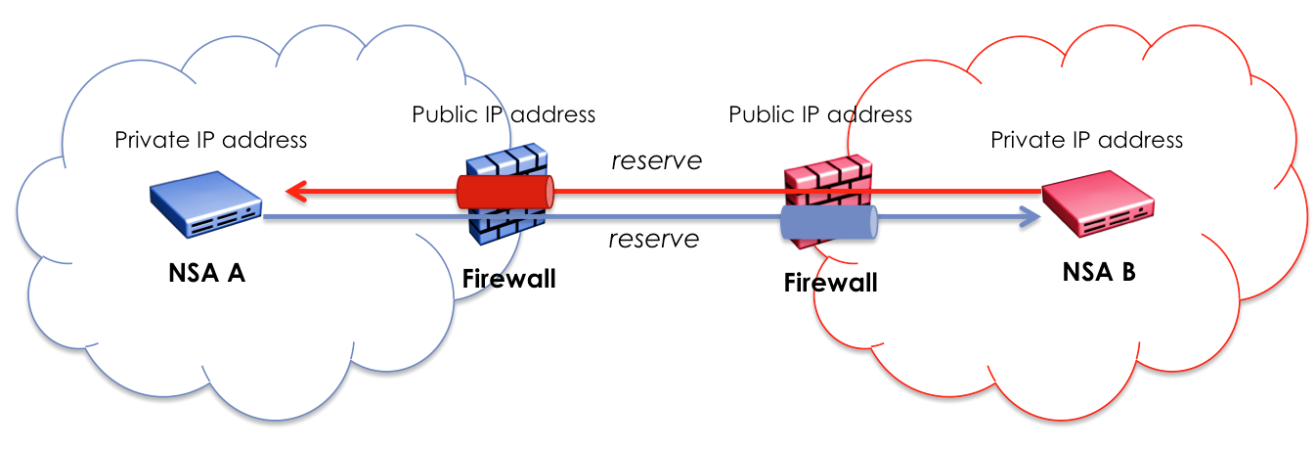
It should also be noted that if these NSA are in a true peer-to-peer configuration both supporting the requester and provider roles, then communications between the two NSA needs to be possible for either NSA to issue requests or return asynchronous confirmations. This also needs to be possible if both NSA are behind firewall devices.

There are a number of solutions to help address these firewall issues. The most obvious is proper firewall configuration for the specific NSA deployment. For an NSA with public IP addresses assigned but behind a firewall, access control lists can be set in combination with port filtering to allow communication between these peer NSAs. This will allow the NSA specific HTTP traffic to be passed between servers, and therefore, achieve proper NSI asynchronous protocol behavior.



**Figure 123 – Peer NSA behind a firewall with public IP addresses.**

A slightly more complicated NSA deployment occurs when one or both of the peer NSA are assigned private IP addresses and are behind a firewall. In this situation the NSA will need to use the IP address of the firewall providing HTTP port forwarding or a full HTTP proxy as its public identity. Access control lists can be set for peer NSA in combination with NAT and port forwarding to allow the RA to be mapped through to the PA’s HTTP server port within the DMZ. However, the key configuration change is a RA behind the firewall will need to provide the public facing IP address and port of the firewall/proxy within the *replyTo* field of the NSI operation request. This will allow the PA to correctly map the SOAP endpoint for the asynchronous response back to the firewall/proxy that will tunnel the message through to the target RA.



**Figure 124 – Peer NSA behind a firewall with private IP addresses.**

To summarize, a PA needs to have a publically accessible interface to receive request messages from an RA, and a RA needs to also have a publically accessible interface to receive response messages (confirm, failed, or event) from the PA.

# Appendix B: Formal statement of coordinator

The following is an attempt to describe the behavior of the Coordinator in relation to the processing of requests and interactions with the various state machines in the NSA. Due to the slight difference in behavior between an AG and uPA, they are described separately.

## Aggregator NSA

### Processing of NSI Requests

**NSI\_rsv.rq(Conn\_ID, Corr\_ID, Ver) */\* from parent NSA \*/***

if (new Conn\_ID) then

{

create state machines RSM(Conn\_ID) /\* initial state = Create Reservation \*/

create state machine, PSM(Conn\_ID), LSM(Conn\_ID) /\* initial state = Created \*/

do pathfinding -> create entry for all children in

connection\_segment\_list(Conn\_ID, Child\_NSA)

}

send res.rq(Corr\_ID, Ver) to RSM(Conn\_ID)

**NSI\_rsvcommit.rq(Conn\_ID, Corr\_ID, ver) */\* from parent NSA \*/***

Create state machine PSM(Conn\_ID) /\* initial state = Released \*/

send rsvcommit.rq(Corr\_ID, Ver) to RSM(Conn\_ID)

**NSI\_rsvabort.rq(Conn\_ID, Corr\_ID, ver)** /\* ***from parent NSA*** \*/

send rsvabort.rq(Corr\_ID, Ver) to RSM(Conn\_ID)

**NSI\_prov.rq(Conn\_ID, Corr\_ID)** /\* ***from parent NSA*** \*/

send prov.rq(Corr\_ID) to PSM(Conn\_ID)

**NSI\_rel.rq(Conn\_ID, Corr\_ID)** /\* ***from parent NSA*** \*/

send rel.rq(Corr\_ID) to PSM(Conn\_ID)

**NSI\_term.rq** /\* ***from parent NSA*** \*/

send term.rq(Corr\_ID) to LSM(Conn\_ID)

send term.rq to RSM(Conn\_ID), PSM(Conn\_ID) /\* if RSM and PSM exist \*/

**NSI\_rsv.cf(Conn\_ID, Corr\_ID)** /\* ***from child NSA*** \*/

set request\_segment\_list(Conn\_ID, Child\_NSA, Corr\_ID).Status = replied

if all children in request\_segment\_list(Conn\_ID, Child\_NSA,

Corr\_ID).Status == replied then

{

send res.cf(Corr\_ID, Ver) to RSM(Conn\_ID)

}

**NSI\_rsv.fl(Conn\_ID, Corr\_ID)** /\* ***from child NSA*** \*/

if request\_list(Conn\_ID, Corr\_ID).Status != fail then

{

set request\_list(Conn\_ID, Corr\_ID).Status = fail

send res.fl(Corr\_ID, Ver) to RSM(Conn\_ID)

}

**NSI\_rsvcommit.cf(Conn\_ID, Corr\_ID, Ver)** /\* ***from child NSA*** \*/

set request\_segment\_list(Conn\_ID, Child\_NSA, Corr\_ID).Status = replied

if all children in request\_segment\_list(Conn\_ID, Child\_NSA,

Corr\_ID).Status == replied then

{

send rsvcommit.cf(Corr\_ID, Ver) to RSM(Conn\_ID)

}

**NSI\_rsvcommit.fl(Conn\_ID, Corr\_ID, Ver)** /\* ***from child NSA*** \*/

set request\_segment\_list(Conn\_ID, Child\_NSA, Corr\_ID).Status = replied

if all children in request\_segment\_list(Conn\_ID, Child\_NSA,

Corr\_ID).Status == replied then

{

send rsvcommit.fl(Corr\_ID, Ver) to RSM(Conn\_ID)

}

**NSI\_rsvabort.cf(Conn\_ID, Corr\_ID, Ver)** /\* ***from child NSA*** \*/

set request\_segment\_list(Conn\_ID, Child\_NSA, Corr\_ID).Status = replied

if all children in request\_segment\_list(Conn\_ID, Child\_NSA,

Corr\_ID).Status == replied then

{

send rsvabort.cf(Corr\_ID, Ver) to RSM(Conn\_ID)

}

**NSI\_prov.cf(Conn\_ID, Corr\_ID)** /\* ***from child NSA*** \*/

set request\_segment\_list(Conn\_ID, Child\_NSA, Corr\_ID).Status = replied

if all children in request\_segment\_list(Conn\_ID, Child\_NSA,

Corr\_ID).Status == replied then

{

send prov.cf(Corr\_ID) to PSM(Conn\_ID)

}

**NSI\_rel.cf(Conn\_ID, Corr\_ID)** /\* ***from child NSA*** \*/

set request\_segment\_list(Conn\_ID, Child\_NSA, Corr\_ID).Status = replied

if all children in request\_segment\_list(Conn\_ID, Child\_NSA,

Corr\_ID).Status == replied then

{

send rel.cf(Corr\_ID) to PSM(Conn\_ID)

}

**NSI\_term.cf(Conn\_ID, Corr\_ID)** /\* ***from child NSA*** \*/

set request\_segment\_list(Conn\_ID, Child\_NSA, Corr\_ID).Status = replied

if all children in request\_segment\_list(Conn\_ID, Child\_NSA,

Corr\_ID).Status == replied then

{

send term.cf(Corr\_ID) to LSM(Conn\_ID)

}

### Requests from State Machines

**rsv.rq(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

create entry for all children in request\_segment\_list(Conn\_ID,

Child\_NSA, Corr\_ID)

send NSI\_rsv.rq(Conn\_ID, Corr\_ID, Ver) to children in

connection\_segment\_list(Conn\_ID, Child\_NSA)

**rsvcommit.rq(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

create entry for all children in request\_segment\_list(Conn\_ID,

Child\_NSA, Corr\_ID)

send NSI\_rsvcommit.rq(Conn\_ID, Corr\_ID, Ver) to children in

connection\_segment\_list(Conn\_ID, Child\_NSA)

**rsvabort.rq(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

create entry for all children in request\_segment\_list(Conn\_ID,

Child\_NSA, Corr\_ID)

send NSI\_rsvabort.rq(Conn\_ID, Corr\_ID, Ver) to children in

connection\_segment\_list(Conn\_ID, Child\_NSA)

**rsv.cf(Corr\_ID)** /\* ***from RSM(Conn\_ID)*** \*/

send NSI\_rsv.cf(Conn\_ID, Corr\_ID, Ver) to the parent

**rsv.fl(Corr\_ID)** /\* ***from RSM(Conn\_ID)*** \*/

send NSI\_rsv.fl(Conn\_ID, Corr\_ID, Ver) to the parent

**rsvcommit.cf(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

send NSI\_rsvcommit.cf(Conn\_ID, Corr\_ID, Ver) to the parent

**rsvcommit.fl(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

send NSI\_rsvcommit.fl(Conn\_ID, Corr\_ID, Ver) to the parent

**rsvabort.cf(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

send NSI\_rsvabort.cf(Conn\_ID, Corr\_ID, Ver) to the parent

**prov.rq(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

create entry for all children in request\_segment\_list(Conn\_ID,

Child\_NSA, Corr\_ID)

send NSI\_prov.rq(Conn\_ID, Corr\_ID) to children in

connection\_segment\_list(Conn\_ID, Child\_NSA)

**rel.rq(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

create entry for all children in request\_segment\_list(Conn\_ID,

Child\_NSA, Corr\_ID)

send NSI\_prov.rq(Conn\_ID, Corr\_ID) to children in

connection\_segment\_list(Conn\_ID, Child\_NSA)

**prov.cf(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

send NSI\_prov.cf(Conn\_ID, Corr\_ID) to the parent

**rel.cf(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

send NSI\_rel.cf(Conn\_ID, Corr\_ID) to the parent

**term.rq(Corr\_ID)** /\* ***from LSM(Conn\_ID)*** \*/

create entry for all children in request\_segment\_list(Conn\_ID,

Child\_NSA, Corr\_ID)

send NSI\_term.rq(Conn\_ID, Corr\_ID) to children in

connection\_segment\_list(Conn\_ID, Child\_NSA)

**term.cf(Corr\_ID)** /\* ***from LSM(Conn\_ID)*** \*/

clean up everything related to Conn\_ID

send NSI\_term.cf(Conn\_ID, Corr\_ID) to the parent

## Ultimate PA

### Processing of NSI Requests

**NSI\_rsv.rq(Conn\_ID, Corr\_ID)** /\* ***from parent NSA*** \*/

if (new Conn\_ID) then

{

create state machines RSM(Conn\_ID), PSM(Conn\_ID), LSM(Conn\_ID)

}

send res.rq(Corr\_ID, Ver) to RSM(Conn\_ID)

if reservation is made by checking the Reservation DB then

{

send res.cf(Corr\_ID, Ver) to RSM(Conn\_ID)

}

else

{

send res.fl(Corr\_ID, Ver) to RSM(Conn\_ID)

}

**NSI\_rsvcommit.rq(Conn\_ID, Corr\_ID, Ver)** /\* ***from parent NSA*** \*/

send rsvcommit.rq(Corr\_ID, Ver) to RSM(Conn\_ID)

**NSI\_rsvabort.rq(Conn\_ID, Corr\_ID, Ver)** /\* ***from parent NSA*** \*/

send rsvabort.rq(Corr\_ID, Ver) to RSM(Conn\_ID)

**NSI\_prov.rq(Conn\_ID, Corr\_ID)** /\* ***from parent NSA*** \*/

send prov.rq(Corr\_ID) to PSM(Conn\_ID)

**NSI\_rel.rq(Conn\_ID, Corr\_ID)** /\* ***from parent NSA*** \*/

send rel.rq(Corr\_ID) to PSM(Conn\_ID)

**NSI\_term.rq(Conn\_ID, Corr\_ID)** /\* ***from parent NSA*** \*/

send term.rq(Corr\_ID) to LSM(Conn\_ID)

send term.rq to RSM(Conn\_ID), PSM(Conn\_ID), ASM(Conn\_ID)

/\* if RSM, PSM and ASM exist \*/

### Requests from State Machines

**rsv.rq(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

ignore

**rsvcommit.rq(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

ignore

**rsvabort.rq(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

ignore

**rsv.cf(Corr\_ID)** /\* ***from RSM(Conn\_ID)*** \*/

set REPLIED(Corr\_ID)

send NSI\_rsv.cf(Conn\_ID, Corr\_ID, Ver) to the parent

**rsv.fl(Corr\_ID)** /\* ***from RSM(Conn\_ID)*** \*/

set REPLIED(Corr\_ID)

send NSI\_rsv.fl(Conn\_ID, Corr\_ID) to the parent

**rsvcommit.cf(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

commit the reservation(Conn\_ID, Ver)

set REPLIED(Corr\_ID)

send NSI\_rsvcommit.cf(Conn\_ID, Corr\_ID, Ver) to the parent

**rsvcommit.fl(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

commit the reservation(Conn\_ID, Ver)

set REPLIED(Corr\_ID)

send NSI\_rsvcommit.fl(Conn\_ID, Corr\_ID, Ver) to the parent

**rsvabort.cf(Corr\_ID, Ver)** /\* ***from RSM(Conn\_ID)*** \*/

abort the reservation(Conn\_ID, Ver)

set REPLIED(Corr\_ID)

send NSI\_rsvabort.cf(Conn\_ID, Corr\_ID, Ver) to the parent

**prov.rq(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

set prov\_flag(Conn\_ID)

if in\_period\_flag is set then

{

activate data plane according to the latest reservation

send prov.cf(Corr\_ID) to PSM(Conn\_ID)

}

**rel.rq(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

reset prov\_flag(Conn\_ID)

deactivate data plane

send rel.cf(Corr\_ID) to PSM(Conn\_ID)

**prov.cf(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

send NSI\_prov.cf(Conn\_ID, Corr\_ID) to the parent

**rel.cf(Corr\_ID)** /\* ***from PSM(Conn\_ID)*** \*/

send NSI\_rel.cf(Conn\_ID, Corr\_ID) to the parent

**term.rq(Corr\_ID)** /\* ***from LSM(Conn\_ID)*** \*/

ignore

**term.cf(Corr\_ID)** /\* ***from LSM(Conn\_ID)*** \*/

clean up everything related to Conn\_ID

send NSI\_term.cf(Conn\_ID, Corr\_ID) to the parent

# References

1. OGF GFD.173: “Network Service Framework v1.0”, <http://www.gridforum.org/documents/GFD.173.pdf>
2. OGF GWD-I Network Service Interface Topology Service Distribution Mechanisms

<https://redmine.ogf.org/dmsf_files/12980?download>=

1. GWD-R-P Network Service Interface Topology Representation
2. OGF GFD.206: Network Markup Language Base Schema version 1 <http://www.gridforum.org/documents/GFD.206.pdf>
3. Network Markup Language Base Schema version 1
4. IETF RFC 5905, Network Time Protocol Version 4: Protocol and Algorithms Specification
5. IETF RFC 4122, A Universally Unique IDdentifier (UUID) URN Namespace
6. ITU-T Rec. X.667 Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components
7. ISO/IEC 9834-8:2005 Information technology -- Open Systems Interconnection -- Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components
8. IETF RFC 4655, "A Path Computation Element (PCE)-Based Architecture", http://www.rfc-editor.org/rfc/rfc4655.txt
9. ISO 8601:2000 “Data elements and interchange formats — Information interchange — Representation of dates and times” or xsd dateTime
10. IETF RFC 5905, “Network Time Protocol Version 4: Protocol and Algorithms Specification”, http://tools.ietf.org/html/rfc5905
11. IETF RFC 6453, “A URN Namespace for the Open Grid Forum (OGF)”, <http://tools.ietf.org/html/rfc6453>
12. OGF GFD-CP.191 "Procedure for Registration of Subnamespace Identifiers in the URN:OGF Hierarchy”, http://www.ogf.org/gf/docs/
13. W3C XML “Schema Definition Language (XSD) 1.1 Part 2: Datatypes”, http://www.w3.org/TR/xmlschema11-2/#anyURI

1. See section 3.3.3 for explanation of the tree hierarchy. [↑](#footnote-ref-1)
2. There is a special case in this version of the NSI CS protocol that requires the PA to retain the “repyTo” field supplied in the reserve request for the duration of the reservation. All other “replyTo” values can be discarded after the confirmed or failed has been delivered to the RA. [↑](#footnote-ref-2)