GFD-R-P.212
NSI-WG

Guy Roberts
Tomohiro Kudoh
Inder Monga
Jerry Sobieski
John MacAuley
Chin Guok

NSI Connection Service v2.0

Status of This Document

Grid Forum Document (GFD).

Copyright Notice

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 (NSI).

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 the Network Services Framework v2.0.

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 4
   1.1 The Connection Service 4
2. Network Service Framework 4
   2.1 NSI Services 4
   2.2 NSI Interface, Agents and Architecture 4
   2.3 NSI Topology 4
   2.4 NSI Service Definitions 5
3. NSI Topology 5
   3.1 Connections and Topology 5
   3.2 Explicit Routing Object 6
   3.3 STP Semantics 6
4. NSI CS messages and state machines 7
   4.1 NSI Messages and operations 7
   4.2 Optional release/provision/modify functionality 10

4.3 NSI state machines 10
4.3.1 Reservation State Machine 11
4.3.2 Provisioning State Machine 13
4.3.3 Lifecycle State Machine 13
4.4 Data Plane Activation 14
4.5 Provisioning Sequence 15
4.6 Guardbands 17
5. NSI Message Transport and Sync/Async messaging 17
5.1 Asynchronous Messaging 17
5.2 Synchronous Messaging 19
5.3 Message format and handling 21
5.3.1 Standard Compliance 21
5.3.2 Message checks 22
5.3.3 ACK handling 22
6. NSI Process Coordination 22
6.1 The Coordinator 22
6.1.1 Communications 23
6.1.2 Per Request Information Elements 23
6.1.3 Correlation Ids and Failure Recovery 23
6.1.4 Information maintained by the Coordinator 25
6.1.5 Per Reservation Information Elements 26
6.1.6 Reservation Versioning Information 26
6.1.7 Data Plane Status Information 27
7. Service Definitions 28
7.1 Context 28
7.2 Service Definitions 28
7.3 Using Service Definitions 29
7.3.1 Providers agree on a common multi-domain service 29
7.3.2 Building an XML Service Definition instance 29
7.3.3 Using SDs to request a service instance 30
7.3.4 Interpreting an incoming request 30
7.4 Service Definitions and a Request workflow 31
8. XML Schema Definitions 32
8.1 NSI CS Versioning 32
8.2 *nsiHeader* element 33
8.2.1 *sessionSecurityAttr* Element 35
8.3 Common types 36
8.3.1 *ServiceExceptionType* 36
8.3.2 *VariablesType* 37
8.3.3 *TypeValuePairType* 37
8.3.4 *TypeValuePairListType* 38
8.3.5 *ConnectionIdType* 38
8.3.6 *DateTimeType* 38
8.3.7 *NsaIdType* 39
8.3.8 *UuidType* 39
8.4 NSI CS operation-specific type definitions. 39
8.4.1 *reserve* message elements 39
8.4.2 *reserveCommit* message elements 42
8.4.3 *reserveAbort* message elements 45
8.4.4 *reserveTimeout* message elements 46
8.4.5 *provision* message elements 48
8.4.6 *release* message elements 49
8.4.7 *terminate* message elements 51
8.4.8 *error* message elements 52
8.4.9 *errorEvent* message elements 53
8.4.10 *dataPlaneStateChange* message elements 55

8.4.11 *messageDeliveryTimeout* message elements 56
8.4.12 *querySummary* message elements 58
8.4.13 *querySummarySync* message elements 60
8.4.14 *queryRecursive* message elements 61
8.4.15 *queryNotification* message elements 63
8.4.16 *queryNotificationSync* message elements 66
8.4.17 *queryResult* message elements 67
8.4.18 *queryResultSync* message elements 70
8.5 NSI CS specific types 73
8.5.1 Complex Types 73
8.5.2 Simple Types 93
9. Security 96
9.1 Transport Layer Security 96
9.2 SAML Assertions 96
10. Contributors 96
11. Glossary 97
12. Intellectual Property Statement 98
13. Disclaimer 99
14. Full Copyright Notice 99
15. Appendix A: State Machine Transition Tables 100
16. Appendix B: Error Messages and Best Practices 101
16.1 Error Messages 101
16.2 NTP servers 102
16.3 Timeouts 102
17. Appendix C: Firewall Handling 103
18. Appendix D: Formal Statement of Coordinator 106
18.1 Aggregator NSA 106
18.1.1 Processing of NSI Requests 106
18.1.2 Requests from State Machines 107
18.2 Ultimate PA 108
18.2.1 Processing of NSI Requests 108
18.2.2 Requests from State Machines 109
19. Appendix E: Service-Specific Schema 110
19.1 Restructuring *criteria* element 110
19.2 The *serviceType* element 110
19.3 Service-specific errors 110
19.4 Point-to-point service-specific schema 111
19.4.1 Service Elements 111
19.4.2 Complex Types 113
19.5 Generic Service Types 114
19.5.1 Complex Types 114
19.5.2 Simple Types 115
19.6 Reservation request 115
19.7 Reservation modification 116
20. Appendix F: Tree and Chain Connection Examples 116
20.1 Connection managed by an NSA chain 116
20.2 Connection managed by an NSA tree 117
21. References 118

1. Introduction

1.1 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 ensure secure service delivery, the NSI Connection Service incorporates 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 be employed over 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-enabled service to identify 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.

2. Network Service Framework

The CS protocol is one of several in the Network Service Interface (NSI) protocol suite; the CS works together with these NSI services to deliver an integrated Network Services Framework (NSF).

The NSI framework and architecture are normatively described in OGF GWD-R-P “Network Service Framework v2.0” [1]. The NSI framework and architecture are summarized here (Section 2) for information purposes only.

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

2.2 NSI Interface, Agents and Architecture

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 basic building block of the NSI architecture is Network Service Agents (NSAs) that communicate using the Network Service Interface (NSI) protocol. The NSI and NSAs exist on the Service Plane. Agents communicate using a flexible hierarchical communication model that allows both tree and chain message delivery models.

2.3 NSI Topology

The NSI extensions [3] to the NML base document [4] describe how NSI Connections are represented using the NSI Topology. This topology representation is based on Service Termination Points (STPs) which are URN identifiers of points were a Connection can be terminated.

2.4 NSI Service Definitions

A Connection request includes service-specific information that describes the requirements of the Connection that is needed. This information will typically include ingress and egress STPs, Explicit Routing Object (ero), capacity of the Connection, and framing information, however the specific information will vary between service types. To allow the new services to be readily defined without a change in the NSI protocol, the service-specific attributes of a Connection request are defined in the documents called the 'Service Definitions'.

A Service Definition is an XML document agreed among the service providers and describes which service parameters can be requested. The Service Definition also includes meta-data that facilitates validation of the requested Connection parameters. So for example, the meta-data defines the range of allowed values for each parameter and whether the parameter is optional or mandatory in a Connection request. Service Definitions are explained in more detail in section 7.

3. NSI Topology

NSI Topology is a topological representation of the service connection capabilities of the network and is used by the NSI CS protocol for resolving service requests. NSI Topology is based on standard NML topology (OGF GFD.206) with NSI specific extensions and constrained naming rules: GWD-R-P Network Service Interface Topology Representation. [3,4]

The NSI Topology exposes a set of Service Termination Point (STP) objects. STPs are used in a Connection request to identify the source, destination and intermediate points of the desired Connection.

3.1 Connections and Topology

Figure 1 shows how NSI Networks interconnect at a shared point known as a Service Demarcation Point (SDP). An SDP is a grouping of two STPs belonging to adjacent connected Networks and is considered to be a virtual point rather than a link.

End-to-end Connections extend across multiple networks; they are constructed by concatenating Connection 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 by concatenating two segments (STP a - STP b) and (STP c - STP d). The inter-Network representation of the Connection (STP a – STP d) maps to a physical instance in the Data Plane.

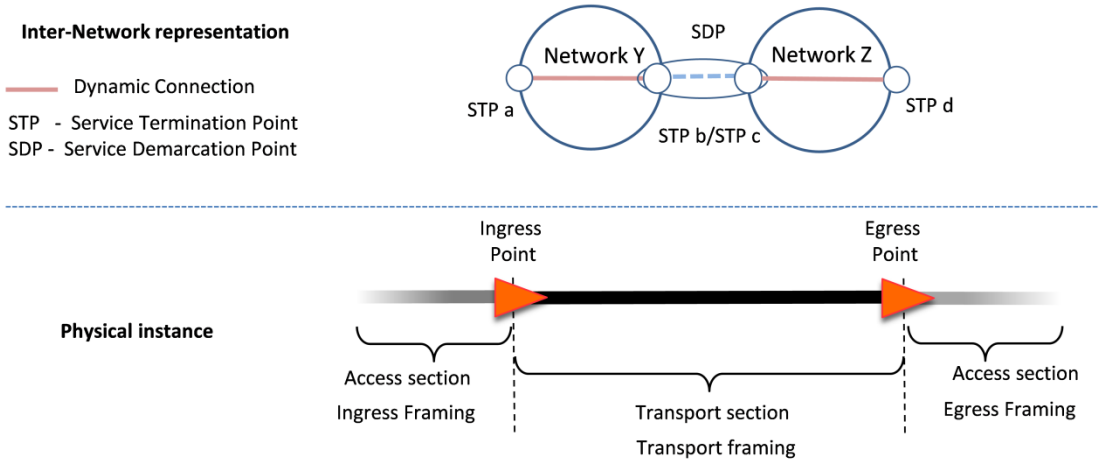

Figure 1: Inter-Network representation of a Connection

3.2 Explicit Routing Object

A Connection request can optionally include an Explicit Routing Object (*ero*) element. An *ero* is an ordered list of STPs that describe the route that should be taken by the Connection. The inter-Network pathfinder will use STPs listed in an *ero* element as constraints during the pathfinding process. The Connection will include all of the STPs in the *ero* in the sequence in which they are listed. However an *ero* is not 'strict' in the sense that a Connection is allowed to transit intermediate STPs between the STPs listed in the *ero*.

Figure 2 shows an example of a Connection. This Connection conforms to any of the following *eros*: (STP b, STP d, STP f), or (STP c, STP e, STP g). Note that as the ingress and egress STPs of a Connection are defined in dedicated fields of the Connection request, they MUST not be included in the *ero*.

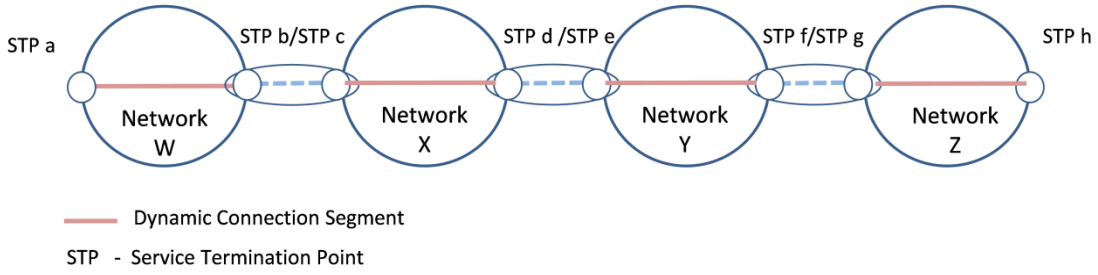

Figure 2: example of an *ero*

The NSI CS does not require NSI messages to be forwarded through the same sequence of NSAs/Networks that the Connection transits, and as a consequence, both tree and chain type architectures are supported. For an example of use of the tree and chain see Appendix F: Tree and Chain Connection Examples.

3.3 STP Semantics

An STP is defined as a three-part identifier comprising a network identifier part, a local identifier part, and a qualifying label part:

```
<STP identifier> ::= <networkId> ":" <localId> <label>
<label> ::= "?" <labelType> "=" <labelValue> | "?"<labelType> | ""
<labelType> ::= <string>
<labelValue> ::= <string>
```

The network identifier points to the domain in which the STP is located, and the local identifier to the specific resource in that domain. The optional label component allows flexibility in STP definition so that the base resource can be identified by the <networkId>:<localId> portion, and then additional qualification by a labelType and/or labelValue pair that can be used to describe technology-specific attributes of the STP (eg. VLAN tags). The labels are defined in NML and for this reason can be interpreted by the Requester Agents (RA) and Provider Agents (PA). . Using these component identifiers makes it possible to easily locate the description of an identifier in the topology. The NSI Topology syntax is normatively defined in the document 'Network Service Interface Topology Representation' [3].

An STP can be fully qualified or under-qualified. A fully qualified STP refers to a specific instance of a resource, (e.g. VLAN or any other element identifiable in NML). An under-qualified STP refers to an STP that is not fully resolved (e.g. it identifies a range of VLANs). Under-qualified STPs are specified using label ranges (e.g. vlan=1780-1790,1799) instead of a single label value.

Both a reserve request and the NSI Topology can make use of under-qualified STPs. The *reserveConfirmed* message MUST return a fully qualified STP, i.e. the NSA must choose one <label> from the list of possible <labels>.

4. 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. The Connection Service includes a set of messages that allow an RA to request connectivity from a PA.

4.1 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.

An asynchronous messaging method has been chosen that supports the indeterminate response times that can arise from complex reservation requests across multiple domains. The NSI CS incorporates an asynchronous callback mechanism permitting unblocking of the CS operation request from the CS confirmed, failed, and error response messages. The RA provides a *replyTo* URL within the NSI header; this URL is then used as the destination of the asynchronous reply.

In addition to asynchronous messaging, the NSI CS supports a limited set of synchronous messages. These have been added specifically to help address the firewall issue described in the appendix. The synchronous messages are based on a simple mechanism that utilizes the basic CS operation request and query messages to provide a functional polling solution.

When asynchronous requests are sent from an RA to a PA, the PA first sends a response for each request, and is then is expected to send an asynchronous reply (confirmed, failed, or error) to each request. When synchronous requests are sent from an RA to a PA, the reply message (confirmed, failed, or error) is included in the response. With SOAP bindings these response messages will be included in the SOAP response part of the SOAP request-response.

The NSI CS message classifications are summarized in Table 1. A list of CS messages from RA to PA is provided in Table 2 and a list of CS messages from PA to RA is provided in Table 3.

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 1 – Message types

Each message invokes a corresponding operation in the recipient by associating it with a message type that can be processed by one of three state machines (See Section 4.3 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 (synchronous or asynchronous).
- 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. Section 8 provides 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 *reserveCommit* message allows an RA to request the PA commit a previously allocated Connection reservation or modify an existing Connection reservation. The combination of the *reserve* and *reserveCommit* are used as a two stage commit mechanism.
reserveAbort (rsvabort.rq)	RSM	Asynch	The *reserveAbort* message allows an RA to request the PA to abort a previously requested Connection that was made using the *reserve* message.
provision (prov.rq)	PSM	Asynch	The *provision* message allows an 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 provisioned Connection into Released state. A Connection in a Released state will deactivate the associated resources in the data plane. The reservation is not affected.
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 an 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 an 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 an 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 an 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.
queryResult ()	Query	Asynch	The *queryResult* message is sent from an RA to a PA to retrieve a list of operation result messages against an existing reservation residing on the PA. A list of operation results will be returned for the specified *connectionId*.
queryResultSync ()	Query	Synch	The *queryResultSync* message is sent from an RA to a PA to retrieve a list of operation result messages associated with a *connectionId* on the PA. Unlike the *queryResult* operation, the *queryResultSync* 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. Section 8 provides a detailed description of these messages and their attributes. Note the *reserveFailed* and *reserveCommitFailed* messages are explicitly required for the state machine.

NSI CS Message (abbreviation)	**SM**	**Synch. /Asynch.**	**Short Description**
reserveResponse ***()***	response	Synch	The *reserveResponse* message is sent to the RA that issued the original *reserve* request immediately after receiving that reservation request to inform the RA of the *connectionId* allocated to that reservation request. There is no impact on the RSM state machine by this message.
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. This response included the summary data requested.
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. This response included the recursive data requested.
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. This response included the summary data requested.
error	error	Asynch	The *error* message is sent from a PA to an RA as an indication of the occurrence of an error condition in response to an original request from the associated RA.
errorEvent ()	notification	Asynch	The *errorEvent* notification is raised when a fault is detected. The message includes attributes that describe the exception and includes the identifier of the NSA generating the exception and the error identifier for each known fault type.
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.
dataPlaneStateChange (*dataPlaneState*Change.nt)	notification	Asynch	The *dataPlaneStateChange* 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 *request* 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. This response includes the summary data requested.
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. This response includes the summary data requested.
queryResultConfirmed ()	query	Asynch	The *queryResultConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *queryResult* request. This response includes the summary data requested.
queryResultSyncConfirmed ()	query	Synch	The *queryResultSyncConfirmed* message is sent to the RA that issued the original request as an indication of a successful operation in response to a *queryResultSync* request. This response includes the summary data requested.

Table 3 – PA to RA Connection Service messages

4.2 Optional release/provision/modify functionality

The release/provision/modify functionality is optionally supported in a PA. To ensure correct transitions of the statemachine, all transitions MUST be carried out as defined in the NSI statemachines regardless of whether the release/provision actions are actually performed.

- Release: If a PA does not support the provision/release cycle on an existing reservation, then the PA MUST spoof a *releaseConfirm* in response to a release request. i.e. a response is returned even though there has been no data-plane affecting changes.
- Provision: PA MUST operate the first provision correctly. If a PA does not support the provision/release cycle on an existing reservation, then the PA MUST spoof a *provisionConfirm* in response to a provision request. I.e a response is returned even though there has been no data-plane affecting changes.
- Modify. If the modify functionality is not supported by a PA, then a *reservedFailed* message MUST be returned with a 'not implemented' error when an attempt is made to modify an existing reservation. When an RA receives a 'not implemented' error, this is a considered a reserve fail event. When the Agg receives a 'not implemented' error, this is forwarded up the tree.

4.3 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 the sequence in which messages are processed. The CS messages are each assigned to one of the three state machines: RSM, PSM and LSM.

When the first *reserve* request for a new Connection is received, the Coordinator MUST coordinate the creation of the RSM, PSM and LSM state machines for that specific connection. For details of the coordinator funcitons see section 6.

The RSM and LSM MUST be instantiated as soon as the first Connection request is received.

The PSM MUST be instantiated as soon as the first version of the reservation is committed.

The following symbols and abbreviations are used in the state machine diagrams.

Abbreviation/symbol	Meaning
Rsv	Reserve
Prov	Provision
Rel	Release
Nt	Notification
Term	Terminate
Rq	Request
Cf	Confirmed
Fl	Failed
>	Downstream input/output
<	Upstream input/output

Table 4 – Abbreviations and symbols used in state machine diagrams

The text boxes show the messages associated with transitions between states. These are color coded as follows:

Red: an input event that is an NSI message – this may be from either a parent or a child NSA.

Blue: an output event that is an NSI message – this is directed towards either a parent or a child NSA.

Appendix A provides a formal statement of the transitions that are allowed in the three state machines.

4.3.1 Reservation State Machine

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

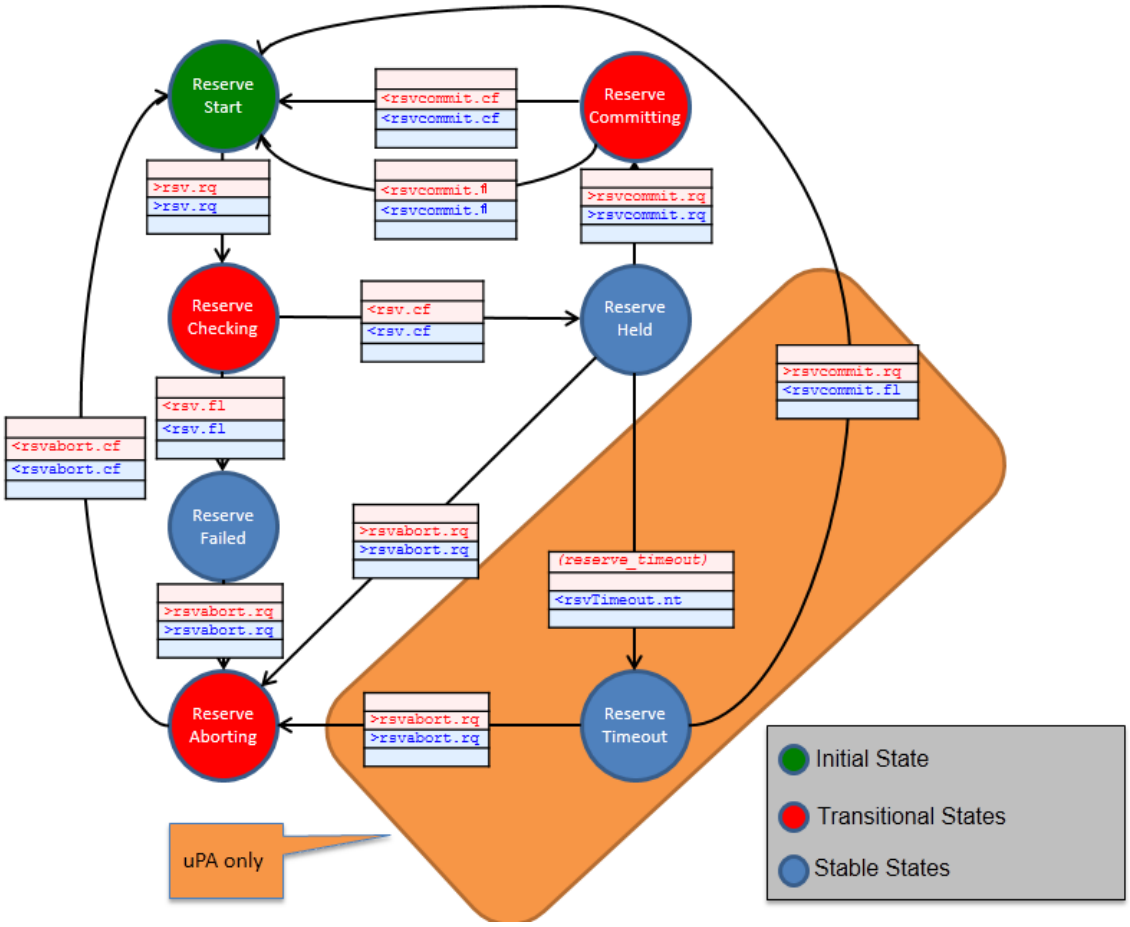

Figure 3: Reservation State Machine

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. The reserveTimeout state is only implemented where the ultimate Provider Agent functionality is present.

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, however, if the initial version of the reservation has not yet been committed, the query will return base reservation information (*connectionId*, *globalReservationId*, *description*, *requesterNSA*, and *connectionStates*) with no versioned reservation criteria. Details of how the version number should be managed can be found in Section 6.1.6.

Modification of start-time, end-time, and service specific parameters are all supported.

4.3.2 Provisioning State Machine

The sequence of operations related to PSM messages MUST conform to the Provision State Machine shown in Figure 3.

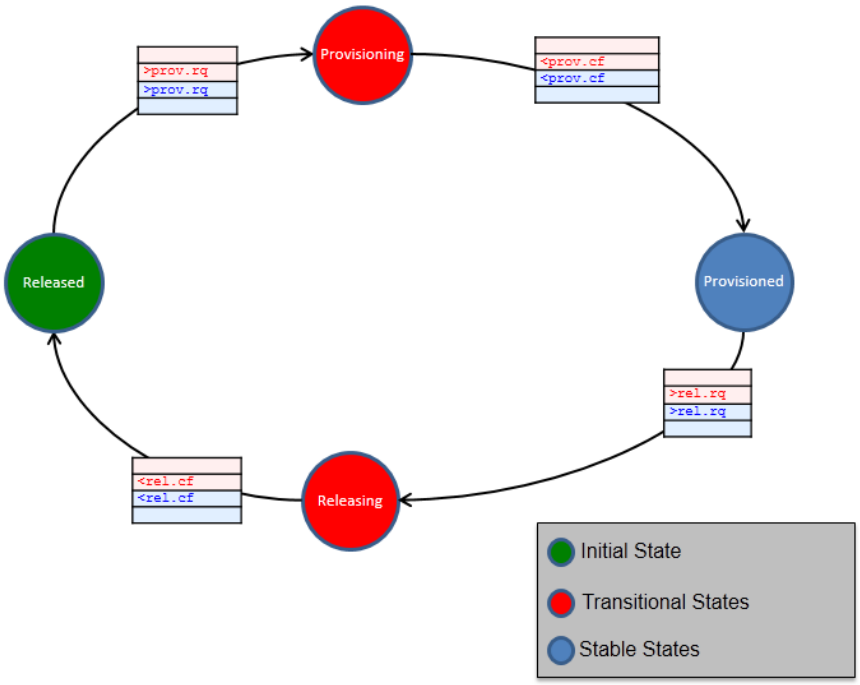

Figure 4: Provision State Machine

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 4.5 for details of the provisioning and activation.

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

4.3.3 Lifecycle State Machine

The sequence of operations related to LSM messages MUST conform to the Lifecycle State Machine shown in Figure 5.

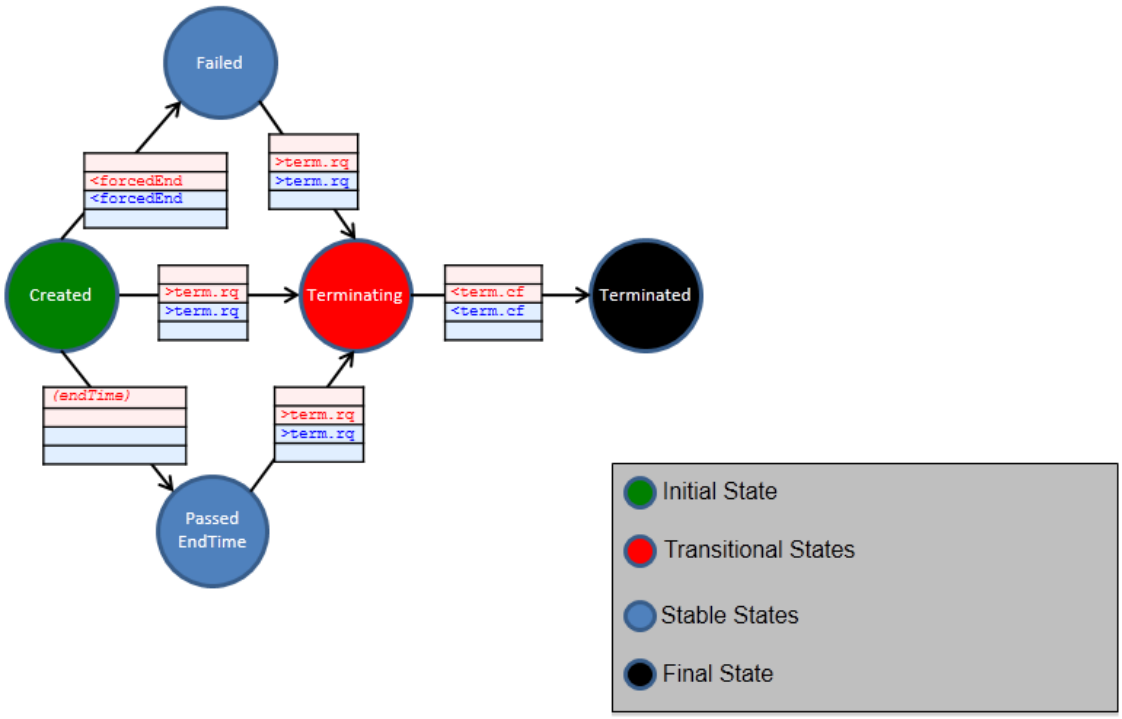

Figure 5: Lifecycle State Machine

The LSM processes *terminate* and *terminateConfirmed* messages. When an *errorEvent* of type *ForcedEnd* is received/sent, the LSM transitions from the ***Created*** to the ***Failed*** state. When current time > end time for the reservation the LSM can be transitioned from ***Created*** to the ***Passed EndTime*** state. The LSM can only transition into the ***Terminated*** stable state through the exchange of *terminate* and *terminateConfirmed* messages.

4.4 Data Plane Activation

Figure 6 below shows the conditions that MUST be met for data plane activation.

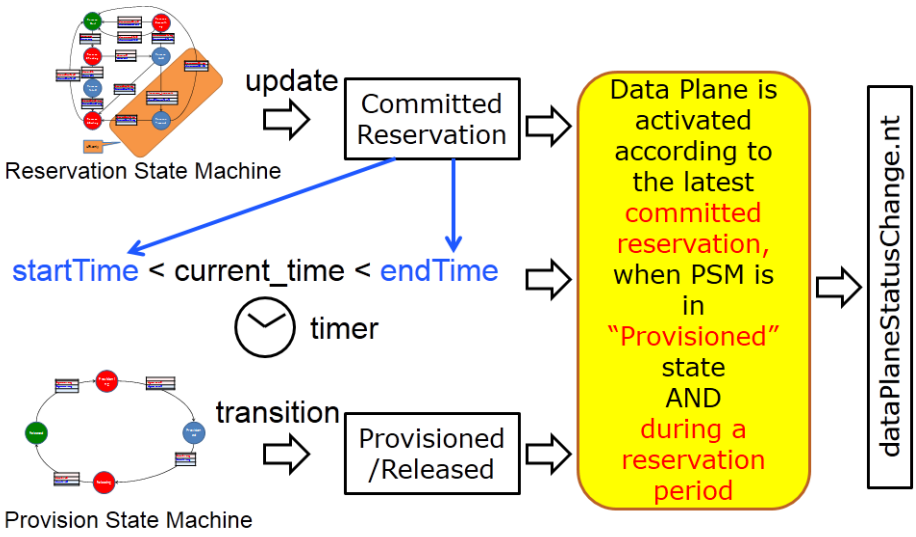

Figure 6: Data Plane activation condition

The Connection can be restored autonomously by the uPA after a failure condition as long as the PSM is in the Provisioned state and current 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.

4.5 Provisioning Sequence

Both automatic and manual provisioning modes MUST be supported. Figure 7 and Figure 8 below show two examples of how message primitives are used to provision and consequently activate a Connection.

Either automatic or manual activation will occur when the conditions described in Figure 6 are met.

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.

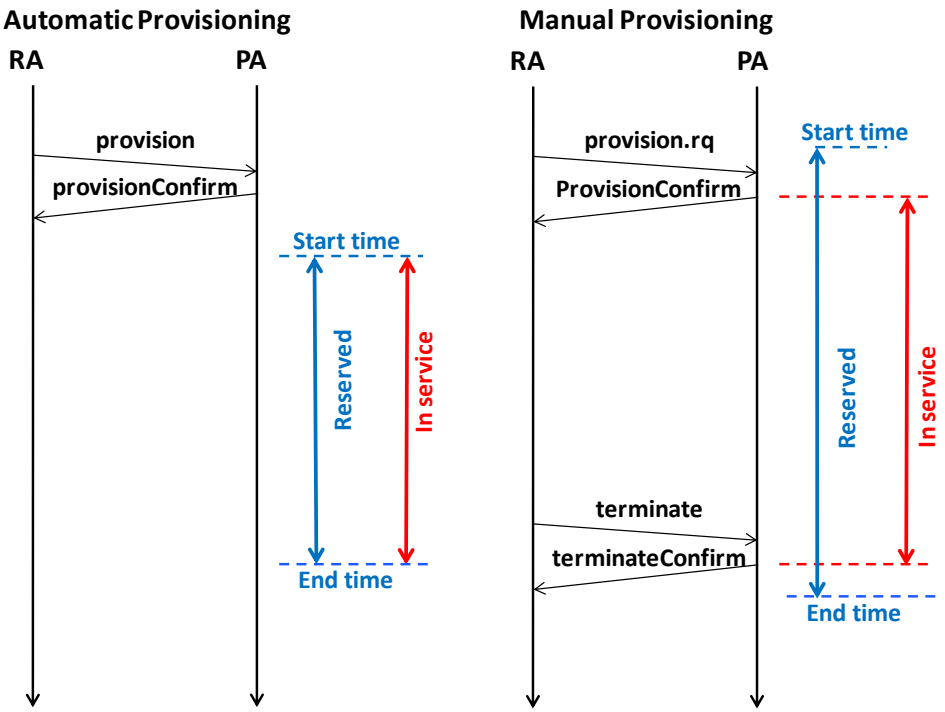

Figure 7: Automatic Provisioning and Manual Provisioning

A Connection can be repeatedly provisioned and released by provision request messages and release request messages, as shown in Figure 8.

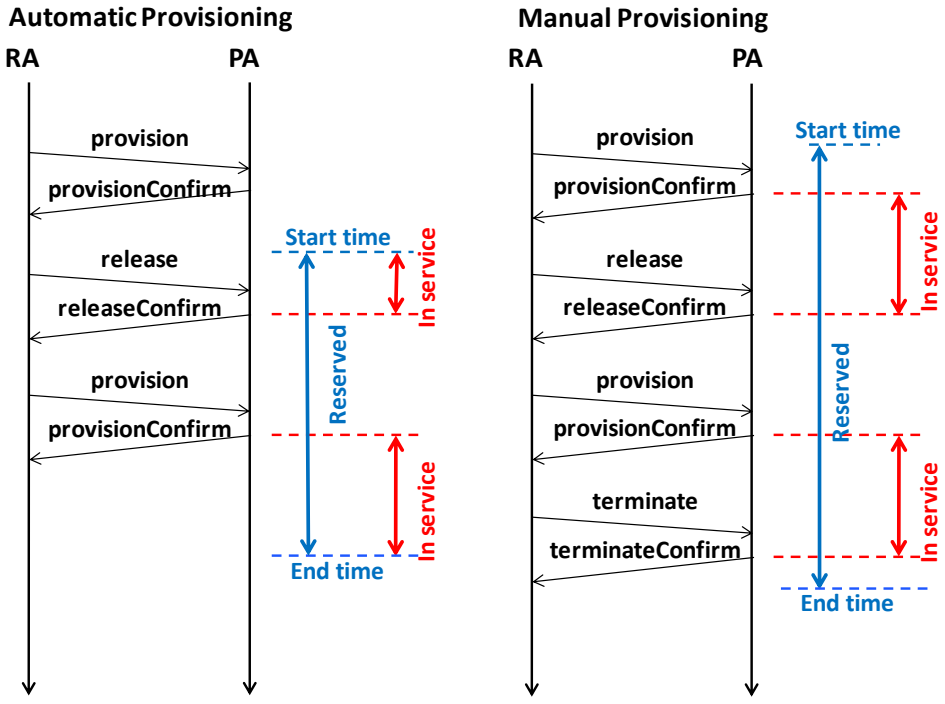

Figure 8: Release and Provisioning

4.6 Guardbands

There may be a delay between the requested in-service start time and the activation of the data plane. So that the RA knows that the data-plane has actually changed state, a state change notification is defined. The dataPlaneStateChange 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.

Start Time. The start time is the earliest that the activation can occur. There may be a delay in completing the activation depending on the time taken by the NRM to perform the activation. In the situation where the RA wishes to ensure that the activation has completed at a guaranteed point in time, it is the responsibility of the RA to add a guard band as they see fit to the start time. The RA is responsible for choosing an appropriate guard time based on their knowledge of the expected provisioning delay at t.he target NRM.

End Time. The end time is the earliest that the deactivation can occur. There may be a delay in completing this action depending on the time taken by the NRM to complete the deactivation. In the situation where the RA wishes to ensure that the deactivation has either started before or completed after at a guaranteed point in time, it is the responsibility of the RA to add a guard band as they see fit to the end time. The RA is responsible for choosing an appropriate guard time based on their knowledge of the expected deactivation delay at the target NRM.

5. NSI Message Transport and Sync/Async messaging

5.1 Asynchronous Messaging

This section describes the messaging interaction models utilized within an NSI CS implementation.

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 be suitable for NSI. For this reason the NSI CS supports 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, NSI defines 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, NSI CS defines a simple mechanism that permits an RA to provide a *replyTo* URL within the NSI header of the operation request message. This URL is a SOAP endpoint that the RA exposes to the PA to receive confirmed, failed, error, and notification messages. 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, failed, or error message to the RA's SOAP endpoint.

Figure 9 shows the basic asynchronous NSI request/reply model. In this case the NSI CS request message is issued from an RA to a PA. If the request is successfully delivered to the PA the MTL layer MUST send an ACK response message immediately after receiving the request to acknowledge to the RA that the request has been accepted by the Coordinator for processing. If an error is detected at this stage, a *serviceException* is returned. The RA will block until either the request's response is received, or an exception is returned. This blocking operation is expected to be extremely short lived as the PA is only acknowledging the acceptance of the request for processing. The MTL MUST provide the ACK response message to the NSA Coordinator.

For the HTTP/SOAP binding the following generic behavior SHOULD be observed for asynchronous messaging:

- 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 *serviceException.*
- The HTTP socket on the RA blocks until the response is returned (Standard HTTP synchronous behaviour).

Sometime later, the PA will have assembled the data requested or determined that the request cannot be satisfied. At this point the PA will make the asynchronous delivery of the reply message back to the RA, as show in the lower half of Figure 9. If the request is successfully delivered to the RA the MTL layer MUST send an ACK response message immediately after receiving the reply to acknowledge to the PA that the confirmed or failed message has been accepted by the Coordinator for processing. If an error is detected at this stage, a *serviceException* is returned. The PA MUST maintain the *repyTo* endpoint value specified in the original operation request until it has delivered a confirmed or failed message back to the RA. The MTL MUST provide the ACK response message to the NSA Coordinator.

For the HTTP/SOAP binding the following generic behavior SHOULD be observed:

- The HTTP POST request carries the NSI CS reply.
- The HTTP 200 OK response carries an acknowledgement indicating successfully delivery of the confirmed message, or a *serviceException* in the case of a processing failure
- The HTTP socket on the PA blocks until the response is returned (Standard HTTP synchronous behaviour).

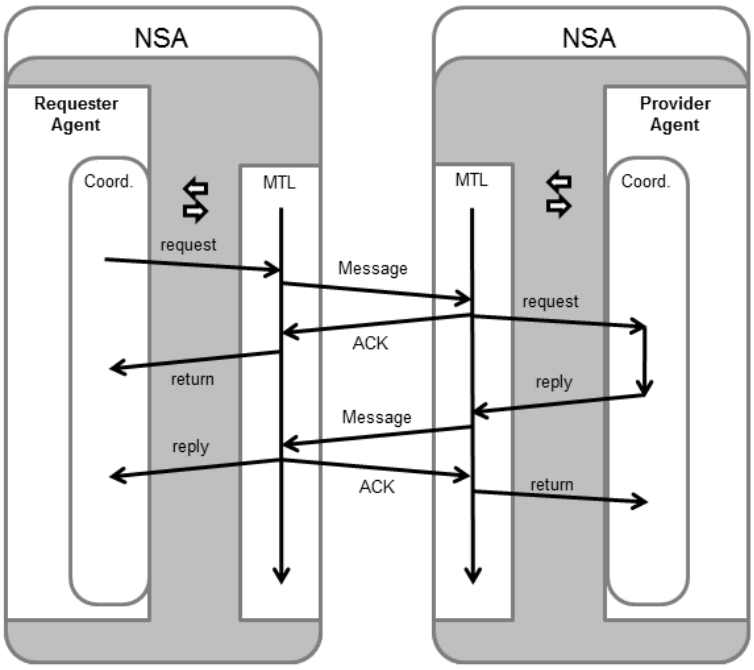

Figure 9: Asynchronous messages and MTL and Coordinator functions.

The asynchronous NSI *reserve* request has some special aspects:

- Instead of the MTL layer sending the generic ACK response message a specific *reserveResponse* message MUST be sent. This message contains the *connectionId* which is assigned by the PA and thus the MTL MUST obtain this from the PA NSA.

- In this version of the NSI CS protocol the PA MUST retain the "*repyTo*" field supplied in the *reserve* request for the duration of the reservation. This "*repyTo*" field SHOULD be used for the notification messages. All other "replyTo" values can be discarded after the confirmed or failed has been delivered to the RA.

Although most NSA deployments will support the described protocol interactions, there are situations where an 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.

5.2 Synchronous Messaging

Figure 10 shows the operation of a synchronous message; an NSI CS request message is issued from the RA, transmitted and received by the MTL layers and passed to the PA for processing. When the PA has collected the required information, or determined that the request cannot be satisfied, this information is sent back to the RA. The RA blocks until the response is returned, and there are no ACK messages involved.

For the HTTP/SOAP binding the following generic behavior SHOULD be observed:

- The HTTP POST request carries the NSI CS operation request with the *replyTo* header element absent.
- The HTTP 200 OK response carries either the requested data or a *serviceException.*
- The HTTP socket on the RA blocks until the response is returned (Standard HTTP synchronous behaviour).

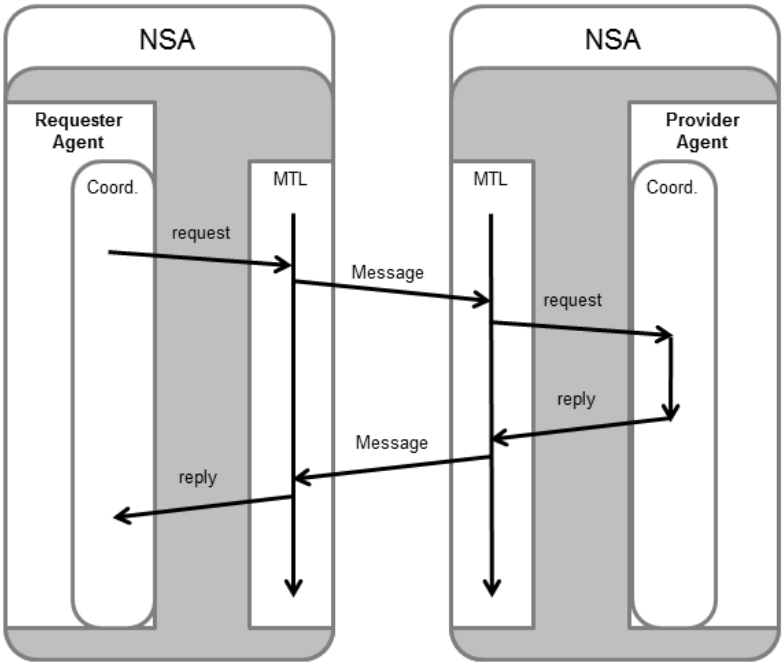

Figure 10: Synchronous messages and MTL and Coordinator functions.

Most NSI messages operate in the asynchronous mode only, however, some messages also support a synchronous mode of operation. This removes the need for asynchronous callbacks for a requester-only NSA. This simple mechanism utilizes the basic CS operation request messages in combination with synchronous version of the query messaged 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.

As indicated in Figure 10 the synchronous messaging model relies on the mechanisms described below to remove the need for asynchronous callbacks, and permit a firewall safe RA implementation:

1. The RA MUST inform 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.
2. If the request is successfully delivered to the PA the MTL layer MUST send an ACK response message immediately after receiving the request to acknowledge to the RA that the request has been accepted by the Coordinator for processing.
3. Note: The *reserve* operation returns the PA allocated *connectionId* for the reservation in the synchronous *reserveResponse* message (this is distinct from the *reserveConfirmed* and *reserveFailed* asynchronous messages).
4. The PA will perform the requested operation, but MUST NOT send a confirmed/failed/error message back to the RA.
5. The RA SHOULD 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. Alternatively, the queryResultSync operation can be used to retrieve any operation result messaged (confirmed, failed, error) generated against the *connectionId*.
6. Notifications generated against a *connectionId* are identified in the reservation query result, and SHOULD be retrieved using the *queryNotificationSync* operation.

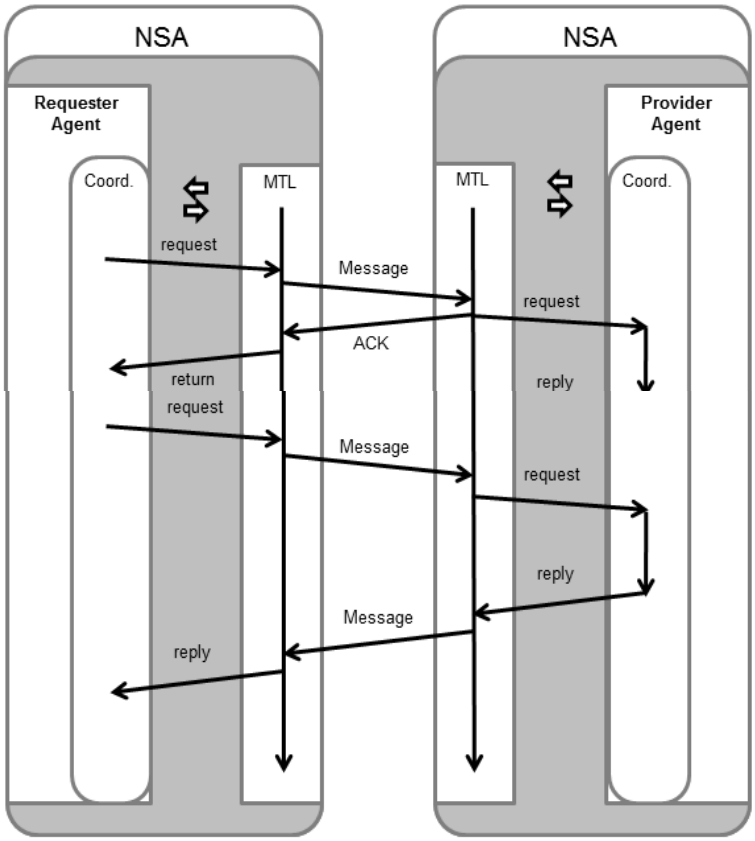

Figure 11: Asynchronous request with synchronous retrieval of the information.

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.

5.3 Message format and handling

5.3.1 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 by the NSI-CS element carried in the SOAP message body. In addition, the the operation is uniquely 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.

5.3.2 Message checks

Additional error condition handling: Received messages must pass the following set of checks in order to be considered valid 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, confirmed, failed, or error message to be returned to the *requesterNSA*. MUST be unique within the context of the *providerNSA* otherwise the request cannot be accepted. See Section 6.1.2 for a description of correlationIds.
- *replyTo* - the confirmed, failed, or error message will be sent back to this location. The contents of the endpoint do not need to be validated, the PA SOULD check the presence of data in the *replyTo* field. The *replyTo* field may left empty to indicate the need to synchronous operation.
- *Reservation* – if the reservation parameters are not present then the message is rejected.
- *requesterNSA* and *providerNSA* – MUST be present for processing to proceed. The *providerNSA* must 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 assigned by the *providerNSA*.
- 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 - "00101", "Invalid or missing parameter" for this generic case and specify attributes identifying the parameter in question. In some cases lower layer errors may mean that it is not possible to send an *NSIServiceException*, in this case a SOAP exception is appropriate.

5.3.3 ACK handling

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

1. For protocol robustness, the NSA SHOULD accept any confirmed/failed messages even if these are received out-of-order with respect to the ACK, i.e. before the associate ACK has been received.
2. The receipt of a confirmed/failed message cancels out the need to receive an ACK. So the NSA should not only continue to process the confirmed/failed 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 confirmed/failed message.
4. TCP will take care of ACK retransmission in case of a packet loss.
5. If the message transport layer is unable to transmit packets, the ACKs will eventually timeout and generate a message transport error that the NSA will need to handle.

6. NSI Process Coordination

6.1 The Coordinator

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

6.1.1 Communications

Reliable communications are 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:

- Whom was the (NSI request) message 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. confirmed, failed, error) for the initial NSI request?

6.1.2 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) that 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 13.), 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 13.).

6.1.3 Correlation Ids and 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 they receive 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. Requesting message redelivery is allowed for all message types.

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 the original message (i.e. Coordinator timeout). 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.

The RA MUST leave the *connectionId* field empty in the initial reservation request.

The workflow in case of resend is shown in Figure 12:

1. NSA-1 (uRA) makes request to NSA-2 (AG) with correlation ID (CorrID) “uRA-1”
2. NSA-2 forward the request to NSA-3 (uPA) with CorrID “AG-1”
3. NSA-2 forward the request to NSA-4 (uPA) with CorrID “AG-2”
4. NSA-3 replies to the request with the corresponding CorrID “AG-1”
5. NSA-2 does not receive a reply from NSA-4, which flags either an MTL timeout (no ACK), or a Coordiinator timeout (no reply)
6. NSA-2 returns an MTL/Coor Timeout error to NSA-1 with the corresponding CorrID “uRA-1” of the initial request
7. NSA-1 decides to resend the initial request for redelivery, which contains the original CorrID “uRA-1” As long as the message transaction remains incomplete all partial messages SHOULD be retained.
8. NSA-2 resends the message to NSA-4 (the only child that was non-responsive) with an initial CorrID “AG-2”
9. NSA-4 replies to the request with the corresponding CorrID “AG-2”
10. NSA-2 aggregates the replies from NSA-3 and NSA-4, and sends the aggregated replyto NSA-1 with the corresponding CorrID “uRA-1”

**NB: If NSA-4 did not receive the initial request from NSA-2 (CorrID = AG-2), NSA-4 will process the request accordingly and return a reply (corrID = AG-2). However if NSA-4 did send a reply to the initial request from NSA-2, but this was not received by NSA-2, then, when NSA-4 receives the “duplicate” request from NSA-2 (CorrID = AG-2), it can simply return the initial reply message (CorrID = AG-2) and not re-process the duplicate request.*

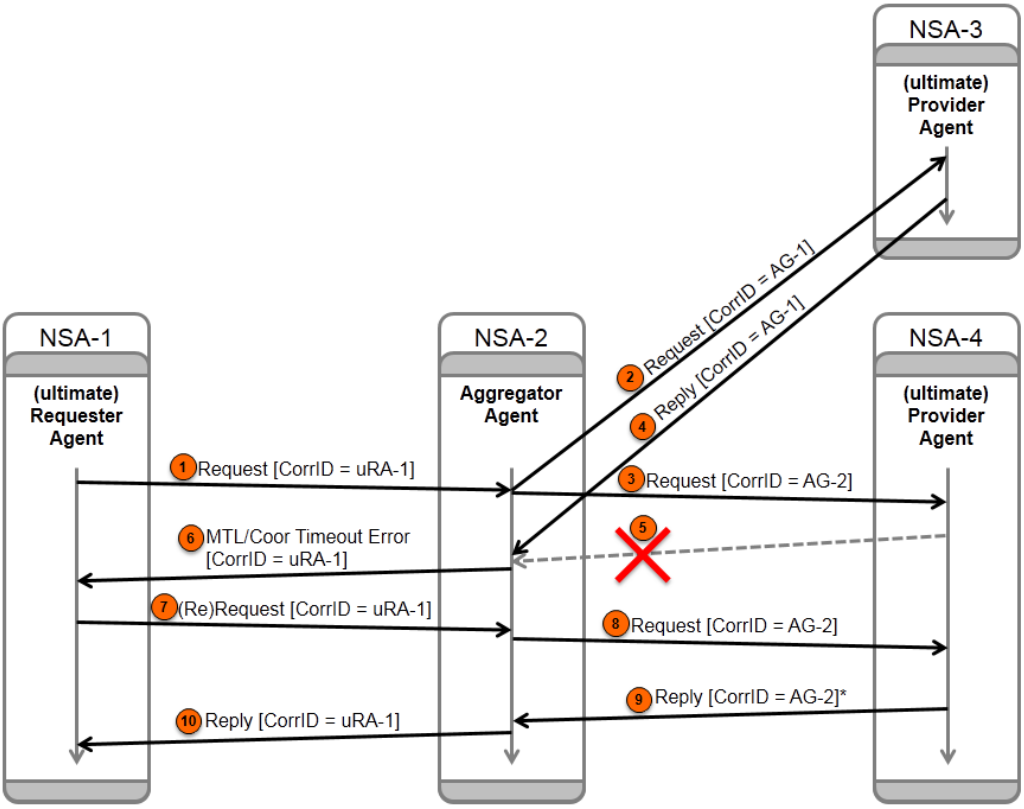

Figure 12: workflow when attempting a message re-send.

6.1.4 Information maintained by the Coordinator

While per request information (see Section 6.1.2 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.

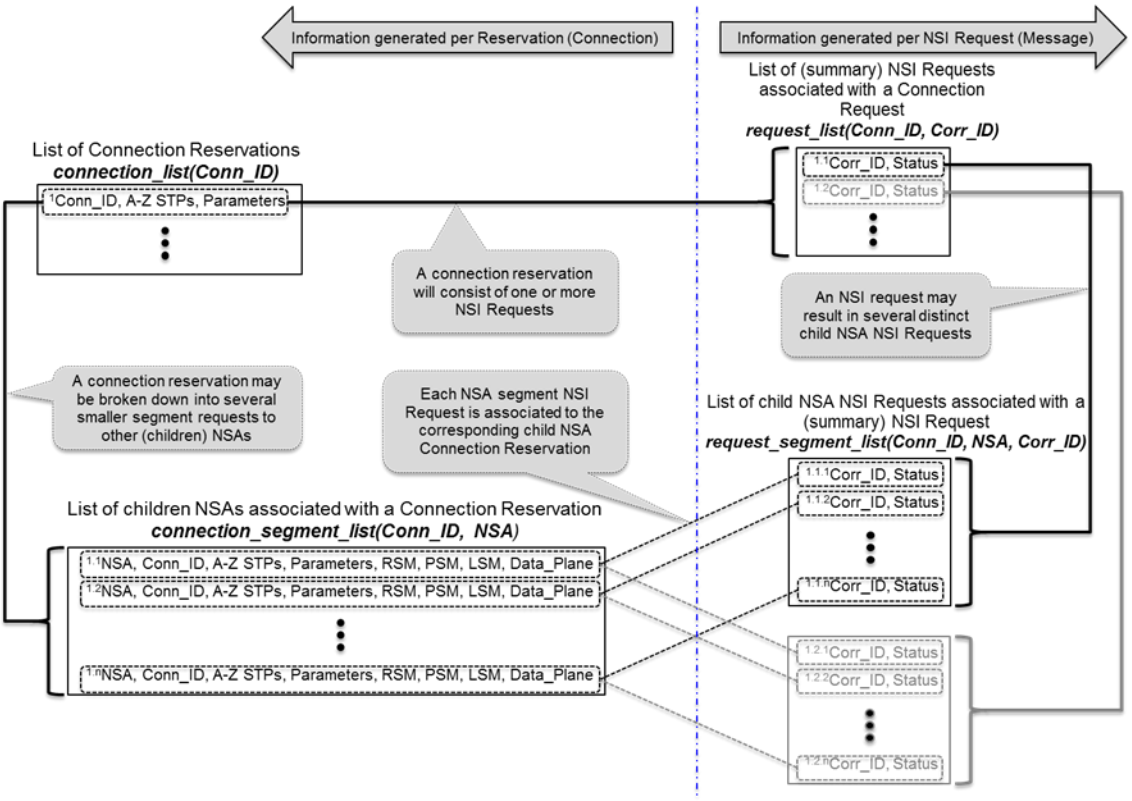

Figure 13: Information maintained by Coordinator for each Connection Reservation and NSI Request

6.1.5 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 13).

- 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* *capacity*, 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 Sections 6.1.6 and 6.1.7 for more details), and if the version is consistent.

6.1.6 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

- If a version number is not specified in an NSI_rsv.rq, it is assumed to be 0 (zero) regardless of whether the request is theinitial or a subsequentrequest.
- 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 6.1.7 Data Plane Status Information)
- If a reservation request attempt fails, or a held initial reservation is aborted and the RSM is in the ReserveStart state, then no version number will be returned.
- 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)
- Versions numbers of failed reservations can be re-used as long as they are numerically higher than the currently committed reservation number

6.1.7 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 of that Connection 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* 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 ChildrenDataPlaneStatus[1..n].Active are true then
{
    DataPlaneStatus.Active = true
```

```
}
else {
    DataPlaneStatus.Active = false
}
DataPlaneStatus.Version = maximum(ChildrenDataPlaneStatus[1..n].Version)
If all ChildrenDataPlaneStatus[1..n].Version are the same, and
   all of ChildrenDataPlaneStatus[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.

The uRA and AG MUST accept *dataPlaneStateChange* notifications associated with a reservation even if they arrive before StartTime. The reaon is that in case there is clock timing issues within network notifications will not be lost.

7. Service Definitions

7.1 Context

In NSI CS version 1.x only unidirectional and bidirectional point-to-point services were offered as part of the protocol. This limitation meant that new service types could not be added without changing the NSI CS schema. This limitation has been removed in NSI CS version 2.0.

Service Definitions are introduced as a mechanism that adds flexibility in the protocol by decoupling the parts of the NSI CS schema used for requesting and provisioning a Connection (the NSI CS base schema) from the schema that describes the requested service and its associated parameters (the service specific schema and Service Definition). This decoupling makes it possible for network providers to define new multi-domain services without modifying the base NSI CS protocol.

7.2 Service Definitions

The Service Definition instance describes the requestable elements associated with a specific inter-Network service, such as Connection capacity and endpoints. The Service Definition (SD) is an XML document that includes:

- Service-specific schema: References to the service-specific schemas associated with the NSI CS reservation request.
- Service parameters: A specification of parameters from the service specific schema such as connection *startTime*, *endTime*, ingress STP, egress STP, capacity, and any restrictions on these values.
- The SD also describes attributes of the service that are not specified in the reservation request but describe features of the service being offered.
- The SD describes service-specific errors and their meanings.

These requestable elements include metadata such as information about their optionality, modifiability, and the range of allowed values for each.

The SD becomes the definitive source of type (via service-specific schema), and units/range definitions for the service. If a service-specific parameter is to be included in a Connection request it MUST also be present in the associated Service Definition. Only parameters that are in the base schema or the nominated Service Definition can be included in a Connection request.

The SD does not explicitly state which STP labels must be present in a *reserveRequest* message for it to be valid for a particular SD. This is necessary since the STP may be opaque (in the case where there is no label) and it will not be possible to interpret whether the STP refers to port, VLAN or something else entirely.

7.3 Using Service Definitions

The requesting agent should select an appropriate SD for their service request. The SD should describe the service that is needed and be available at all of the NSAs participating in the service – otherwise the request will fail. The Provider Agent interprets the incoming Connection request by inspecting the *serviceType* field and uses this to fetch the SD and then interpret the service-specific elements within the request. The elements of this workflow are described next.

7.3.1 Providers agree on a common multi-domain service

The aim of the Service Definition is to allow a federation of network providers to collaborate to define their own service. First the providers have to agree on the inter-Network service that they wish to offer. This implies that the participating providers must agree to honor a minimum level of service functionality in accordance with this agreement. This will ensure that any provider issuing a Connection request in the service area can be confident that the request will be delivered as long as sufficient network capacity is available.

Once the service is agreed, the network providers can either use a pre-defined Service Definition template or build a new Service Definition.

7.3.2 Building an XML Service Definition instance

In many situations it is expected that one of the pre-defined set of SDs (such as the P2P Ethernet VLAN Transfer Service) will be suitable for describing a new service. Where a new service is not fully described by an existing SD, then the providers who have developed the new service will develop a new SD to describe the details of the service, ensuring that all requestable parameters are included and fully defined.

The following figure shows diagrammatically how a SD is developed. The SD is built up by incorporating parameters and attributes from a range of source documents.

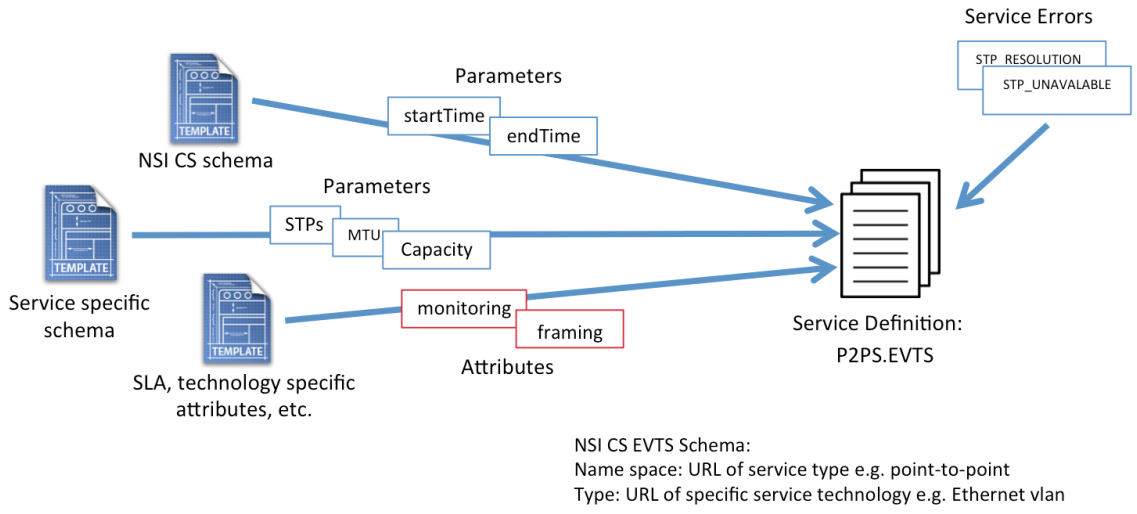

Figure 14: Building a Service Definition

The XML SD instance is built up by incorporating the following elements:

- Parameters from the NSI CS schema (e.g. *startTime*, *endTime*)
- Parameters from the service-specific schema (describes common service-specific types)
- SLA attributes and technology specific attributes (e.g. monitoring, VLAN framing types)
- Service errors (i.e. service errors specific to the service type)

7.3.3 Using SDs to request a service instance

When creating a Connection request message, the elements included in the message will include the CS base schema elements and elements from the appropriate SD. In the example shown below, the specified *serviceType* uniquely identifies an SD instance document requiring that the P2P service element (psp2) must be included in the reservation request. This p2ps element is included as a service specific extension to the existing reservation, with service parameters populated from the P2PServcieBaseType.

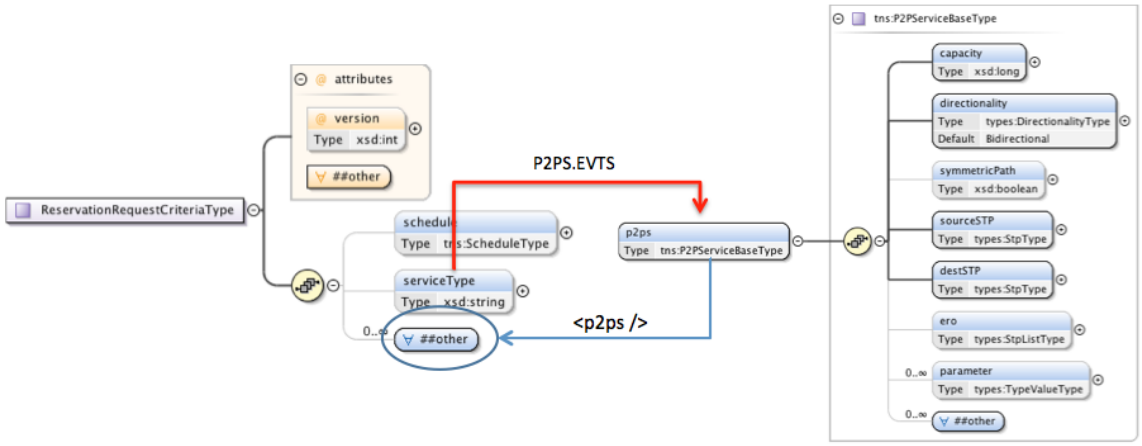

Figure 15: Creating a Connection request using the Service Definition

7.3.4 Interpreting an incoming request

The *serviceType* element relays the specific service type being requested in the reservation. This service type string maps to a specific Service Definition template defined by the network providers describing the type of service offered, parameters supported in a reservation request (mandatory and optional), defaults for parameters if not specified (as well as maximums and minimums), and other attributes relating to the service offering. The NSA in turn uses this information to determine the specific service parameters carried in the criteria element required to specify the requested service.

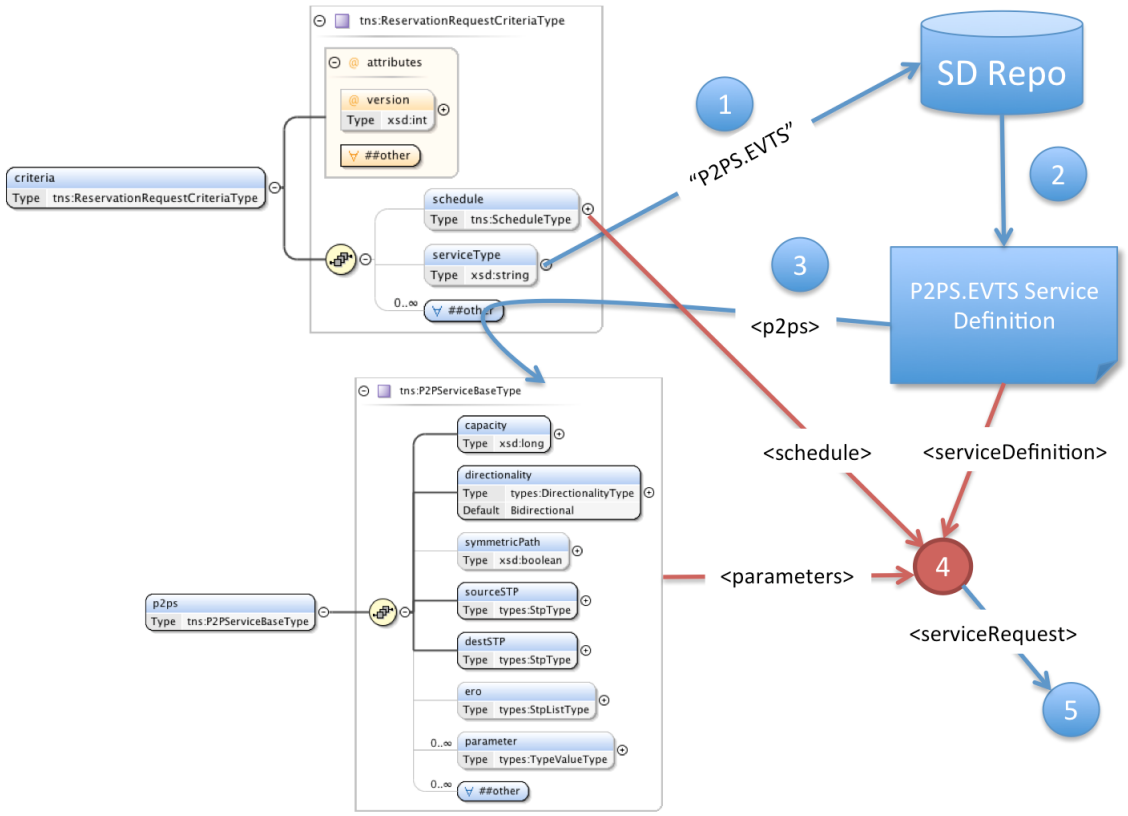

Figure 16: Interpreting a Connection request using the Service Definition

When a *reserveRequest* arrives the following steps are followed:

1. Extract the *serviceType* value.
2. Fetch the Service Definition corresponding to the *serviceType*.
3. Extract the service specific elements from *criteria* as defined in the SD.
4. Use the Service Definition to validate that these parameters are allowed for this service and process the service request using both the supplied service parameters and additional information as needed from the Service Definition document.

7.4 Service Definitions and a Request workflow

The complete workflow for Connection requests is summarized here:

1. The RA enters the parameter values associated with the Connection into the *ConnectionRequest* message, adding service-specific parameters to the *ConnectionRequest* as specified in the SD. Service-specific parameters MUST match the parameters in the SD.
2. The *serviceType* element in the *ConnectionRequest* message MUST identify the SD to which the request is directed.
3. The first NSA to receive the *ConnectionRequest* will parse the request against the nominated SD instance to validate the request.
4. Once validated, the *ConnectionRequest* will then be passed to the path computation element.
5. A successful path computation will result in a Connection being scheduled.
6. If the Connection transits another Network, the new *ConnectionRequest* will use the same SD as the one from the uRA (unless adaptation is performed resulting in a new Connection type).

8. 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/2013/12/framework/types
NSI message header definition.	http://schemas.ogf.org/nsi/2013/12/framework/headers
NSI CS operation-specific type definitions.	http://schemas.ogf.org/nsi/2013/12/connection/types
NSI CS operation definitions	http://schemas.ogf.org/nsi/2013/12connection/interface
PA interface SOAP binding	http://schemas.ogf.org/nsi/2013/12/connection/provider
RA interface SOAP binding	http://schemas.ogf.org/nsi/2013/12/connection/requester

Table 5 – XML namespaces for NSI CS 2.0

8.1 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/2013/12/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 within the NSI suite of protocols utilizes Internet Assigned Numbers Authority (IANA) MIME Media Types as a standard mechanism for distinguishing between releases of each protocol. The current NSI CS 2.0 profile utilizes SOAP over HTTP as a transport that has a standard MIME Media Type of “application/soap+xml”. We have created a custom Media Type for the NSI CS 2.0 SOAP profile to distinguish this protocol, however, it is only used in the *protocolVersion* field of the SOAP header and not the *Content-types* field of the HTTP header that remains “application/soap+xml”.

Table 6 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 8.2).

Version	Interface	MIME Media Type
NSI CS version 1.0	Provider	“application/vnd.ogf.nsi.cs.v1.provider+soap”
NSI CS version 1.0	Requester	“application/vnd.ogf.nsi.cs.v1.requester+soap”
NSI CS version 1.1	Provider	“application/vnd.ogf.nsi.cs.v1-1.provider+soap”
NSI CS version 1.1	Requester	“application/vnd.ogf.nsi.cs.v1-1.requester+soap”
NSI CS version 2.0	Provider	“application/vnd.ogf.nsi.cs.v2.provider+soap”
NSI CS version 2.0	Requester	“application/vnd.ogf.nsi.cs.v2.requester+soap”

Table 6 – NSI CS protocol version MIME Media Types.

8.2 *nsiHeader* element

Namespace definition: http://schemas.ogf.org/nsi/2013/12/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.

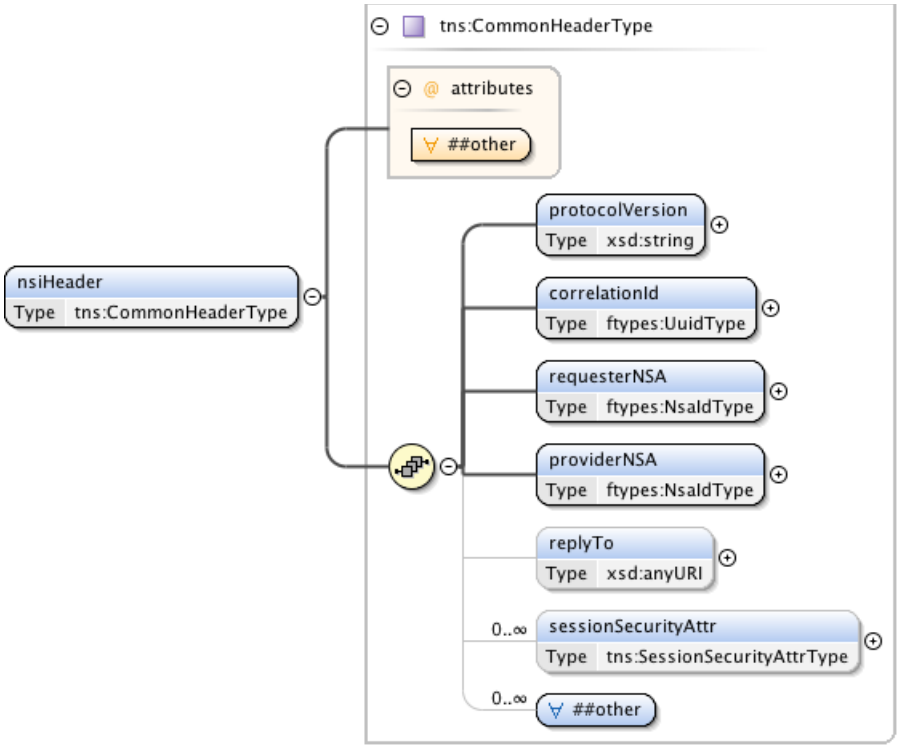

Figure 17 – *nsiHeader* structure.

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 requester 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 7 *nsiHeader* parameters

The following table describes each message and its 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.

M = Mandatory
O = Optional
N/A = Not Applicable

		Header parameters							
		protocolVersion	*correlationId*	*requesterNSA*	*providerNSA*	*replyTo*	*sessionSecurityAttributes*	*other*	*Soapaction*
Messaging Primitives	*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	O	O	M
	reserveConfirmedACK	M	M	M	M	N/A	N/A	O	N/A
	reserveFailed	M	M	M	M	N/A	O	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	O	O	M
	reserveCommitConfirmedACK	M	M	M	M	N/A	N/A	O	N/A
	reserveCommitFailed	M	M	M	M	N/A	O	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	O	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	O	O	M
	provisionConfirmedACK	M	M	M	M	N/A	N/A	O	N/A
	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	O	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	O	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	O	O	M
querySummaryConfirmedACK	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	O	O	M
queryRecursiveConfirmedACK	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
error	M	M	M	M	N/A	O	O	M
errorACK	M	M	M	M	N/A	N/A	O	N/A
errorEvent	M	M	M	M	N/A	O	O	M
errorEventACK	M	M	M	M	N/A	N/A	O	N/A
reserveTimeout	M	M	M	M	N/A	O	O	M
reserveTimeoutACK	M	M	M	M	N/A	N/A	O	N/A
dataPlaneStateChange	M	M	M	M	N/A	O	O	M
dataPlaneStateChangeACK	M	M	M	M	N/A	N/A	O	N/A
messageDeliveryTimeout	M	M	M	M	N/A	O	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	O	O	M
queryNotificationConfirmedACK	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
queryResult	M	M	M	M	M	O	O	M
queryResultACK	M	M	M	M	N/A	N/A	O	N/A
queryResultConfimed	M	M	M	M	N/A	O	O	M
queryResultConfimedACK	M	M	M	M	N/A	N/A	O	N/A
queryResultSync	M	M	M	M	N/A	O	O	M
queryResultSyncConfimed	M	M	M	M	N/A	N/A	O	M

Table 8 – NSI CS message use of header fields

8.2.1 *sessionSecurityAttr* Element

The *sessionSecurityAttr* element is defined using a standardized SAML *AtttributeStatementType* imported from the SAML namespace *urn:oasis:names:tc:SAML:2.0:assertion* with an NSI specific extension to add a string based attribute type and name. This allows for multiple *sessionSecurityAttr* elements to be specified in the header, and each one identified for a specific use (for example, supplying user credentials per NSA domain). The specific use of this element is out of the scope of this document.

The expected (default) behaviour is that an NSA AG MUST pass any received session security attributes on to all children, however, deployment specific behaviours may be introduced that change this default behaviour.

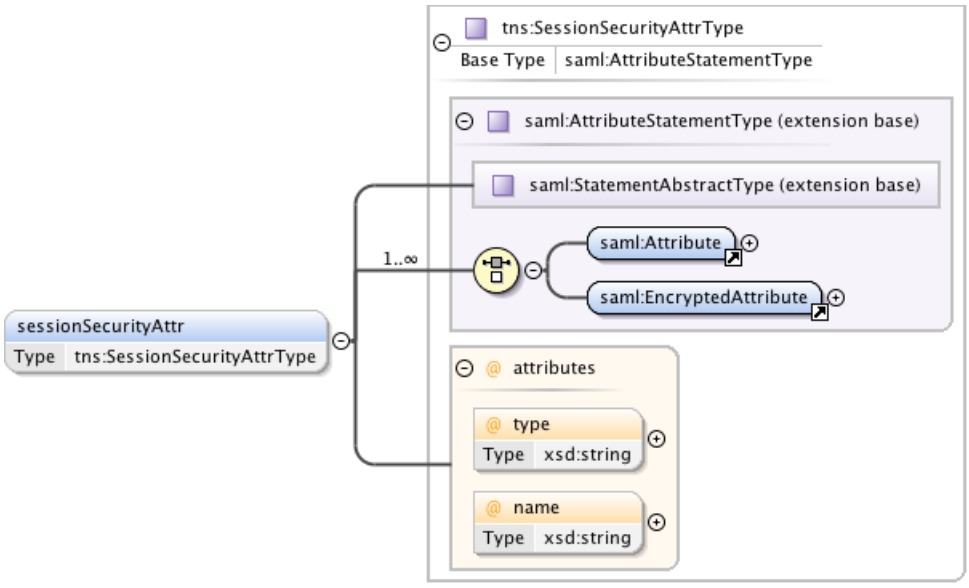

Figure 18 – *sessionSecurityAttr* type.

8.3 Common types

Namespace definition: http://schemas.ogf.org/nsi/2013/12/framework/types

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

8.3.1 *ServiceExceptionType*

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

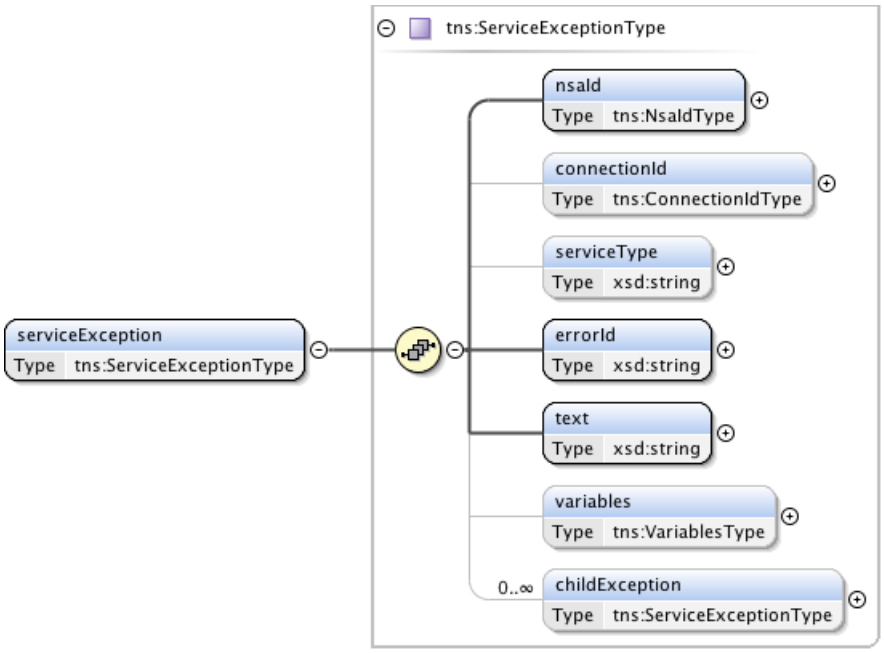

Figure 19 – *ServiceExceptionType* type.

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.
serviceType	O	The service type identifying the applicable service definition within the context of the NSA generating the error.
errorId	M	Error identifier uniquely identifying each known fault within the protocol.
text	M	User-friendly message text describing the error.
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 9 – *ServiceExceptionType* parameters.

8.3.2 *VariablesType*

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

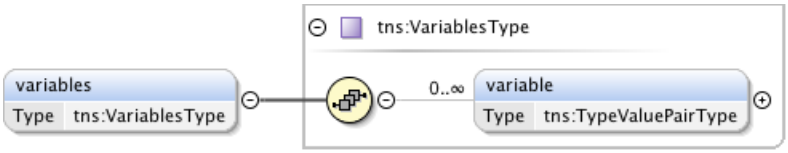

Figure 20 – *NsaIdType* type.

Parameters

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

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

Table 10 – *VariablesType* parameters.

8.3.3 *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.

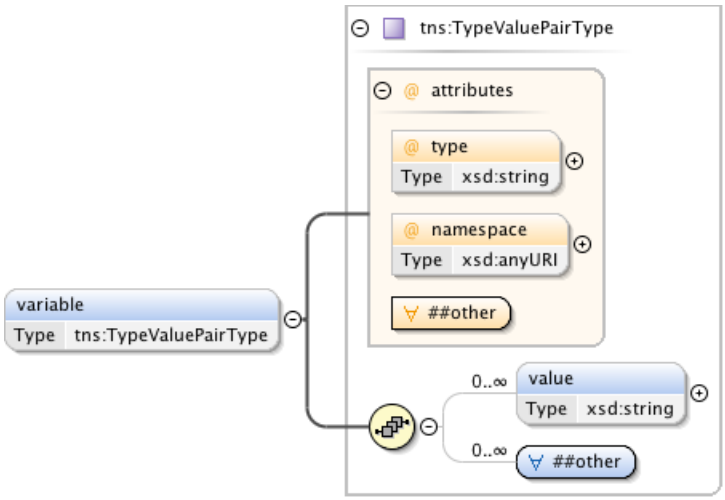

Figure 21 – *TypeValuePairType* type.

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.
namespace	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 11 – *TypeValuePairType* parameters

8.3.4 *TypeValuePairListType*

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

Figure 22 – *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 12 – *TypeValuePairListType* parameters

8.3.5 *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.

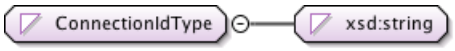

Figure 23 – *ConnectionIdType* type.

8.3.6 *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|[+-].+)"

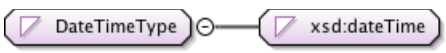

Figure 24 – *DateTimeType* type.

8.3.7 *NsaIdType*

NsaIdType is a specific type for a Network Services Agent (NSA) identifier that is populated with a OGF URN [12], [13] to be used for compatibility with other external systems.

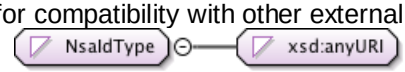

Figure 25 – *NsaIdType* type.

8.3.8 *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 26 – *UuidType* type.

8.4 NSI CS operation-specific type definitions.

Namespace definition: http://schemas.ogf.org/nsi/2013/12/connection/types

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

8.4.1 *reserve* message elements

The *reserve* message is sent from an 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*. The original *connectionId* will be returned for the *reserveResponse* of a modification. 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*
Error	N/A	N/A	N/A	N/A

Table 13 *reserve* message elements

8.4.1.1 Request: *reserve*

The NSI CS *reserve* message allows an RA to reserve network resources associated with a service within the Network constrained by the provided service parameters. This *reserve* message allows an 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 specific *connectionId* or a reservation timeout occurs (whichever arrives first).

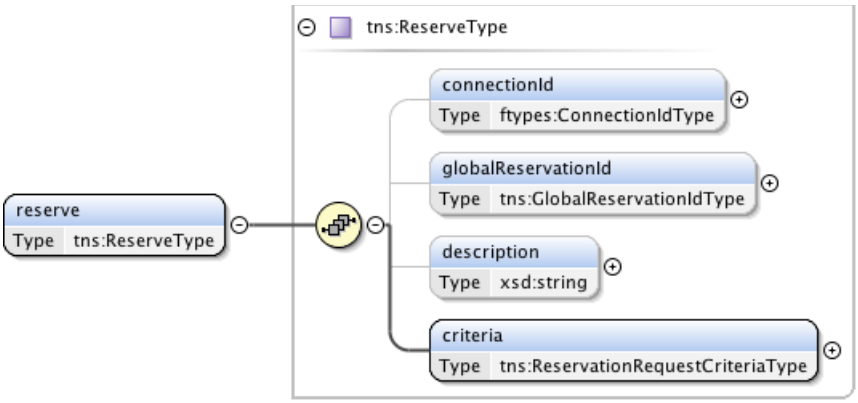

Figure 27 – *reserve* request message structure.

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 version, start and end time, service type, and service-specific schema elements.

Table 14 *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.

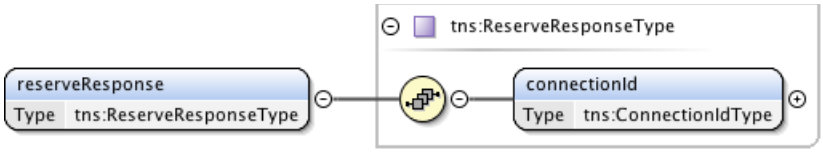

Figure 28 – *reserveResponse* message structure.

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 15 *reserveResponse* message parameters

8.4.1.2 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 a reservation timeout occurs (whichever arrives first).

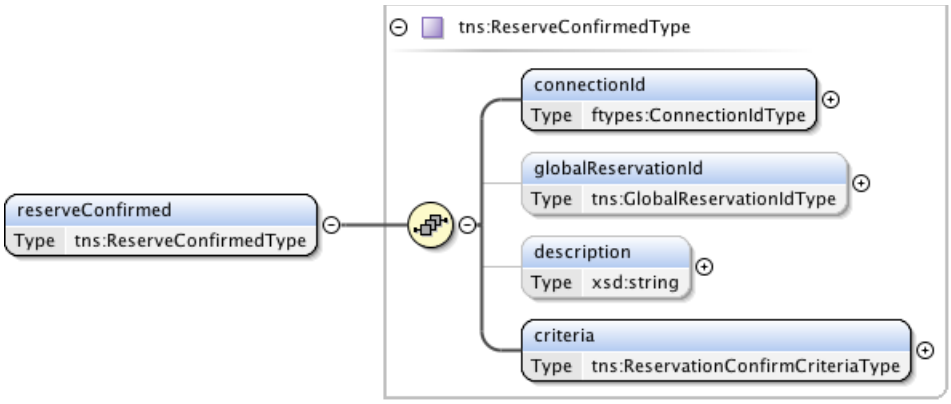

Figure 29 – *reserveConfirmed* message structure.

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 16 *reserveConfirmed* message parameters

Response

If the *reserveConfirmed* operation is successful, a *reserveConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

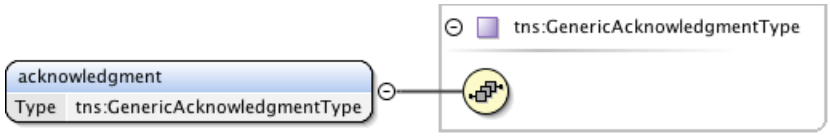

Figure 30 – *reserveConfirmedACK* message structure.

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

8.4.1.3 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.

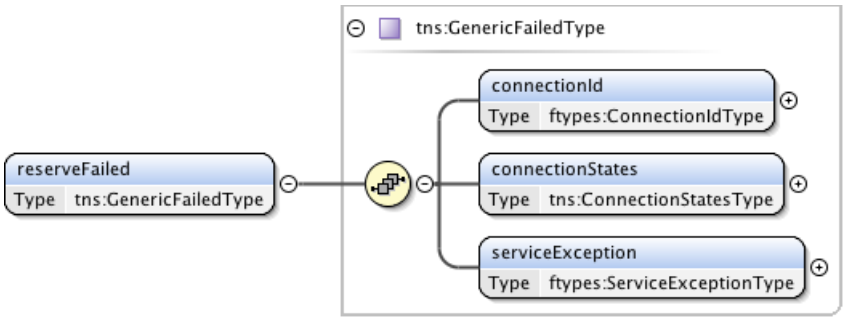

Figure 31 – *reserveFailed* message structure.

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 17 *reserveFailed* message parameters

Response

If the *reserveFailed* operation is successful, a *reserveFailedACK* message is returned, otherwise a *serviceException* is returned. An 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.

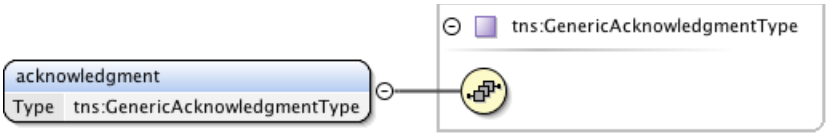

Figure 32 – *reserveFailedACK* message structure.

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

8.4.2 *reserveCommit* message elements

The *reserveCommit* message is sent from an 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*
Error	N/A	*N/A*	*N/A*	*N/A*

Table 18 *reserveCommit* message elements

8.4.2.1 Request: *reserveCommit*

The NSI CS *reserveCommit* message allows an 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.

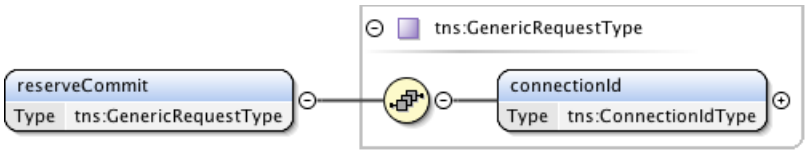

Figure 33 – *reserveCommit* request message structure.

Parameters

The *reserveCommit* message has the following parameters:

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

Table 19 *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.

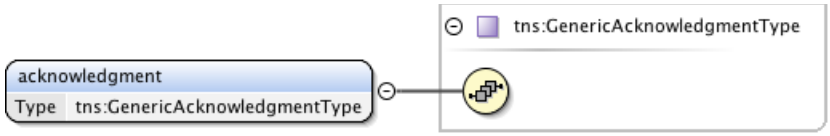

Figure 34 – *reserveCommitACK* message structure.

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

8.4.2.2 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.

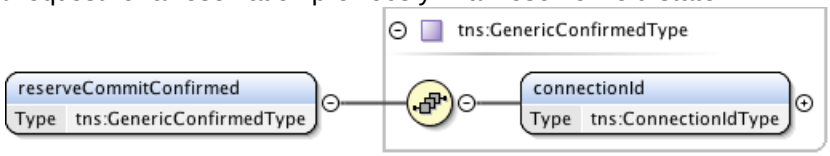

Figure 35 – *reserveCommitConfirmed* message structure.

Parameters

The *reserveCommitConfirmed* message has the following parameters:

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

Table 20 *reserveCommitConfirmed* message parameters

Response

If the *reserveCommitConfirmed* operation is successful, a *reserveCommitConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

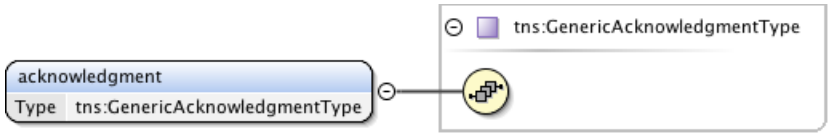

Figure 36 – *reserveAbortConfirmedACK* message structure.

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

8.4.2.3 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.

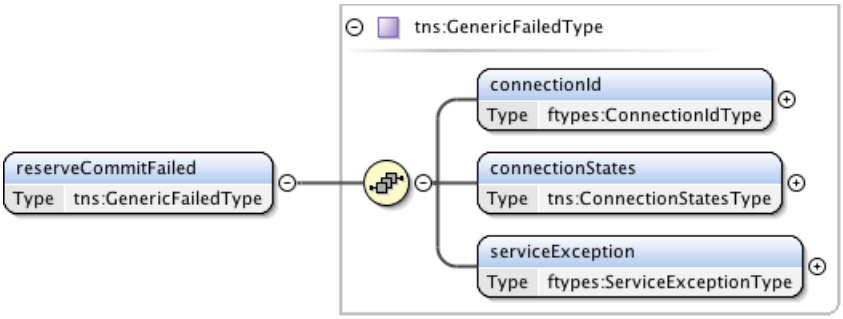

Figure 37 – *reserveCommitFailed* message structure.

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 21 *reserveCommitFailed* message parameters

Response

If the *reserveCommitFailed* operation is successful, a *reserveCommitFailedACK* message is returned, otherwise a *serviceException* is returned. An 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.

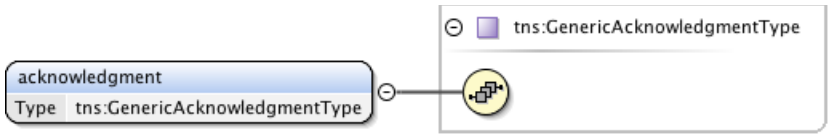

Figure 38 – *reserveCommitFailedACK* message structure.

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

8.4.3 *reserveAbort* message elements

The *reserveAbort* message is sent from an 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
Error	PA to RA	error	errorACK	*serviceException*

Table 22 *reserveCommitFailed* message elements

8.4.3.2 Request: *reserveAbort*

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

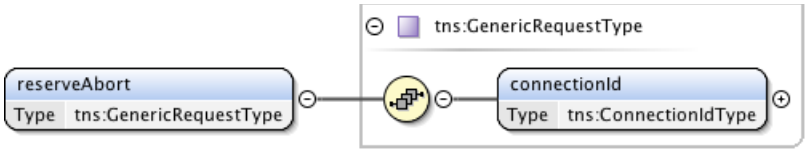

Figure 39 – *reserveAbort* request message structure.

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 23 *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.

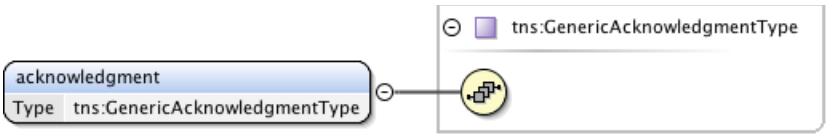

Figure 40 – *reserveAbortACK* message structure.

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

8.4.3.3 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.

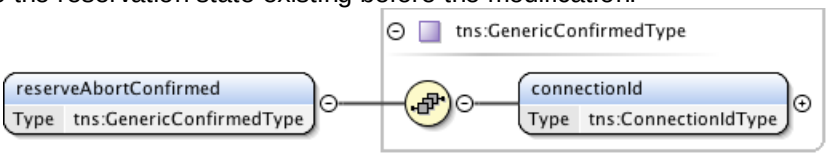

Figure 41 – *reserveAbortConfirmed* message structure.

Parameters

The *reserveAbortConfirmed* message has the following parameters:

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

Table 24 *reserveAbortConfirmed* message parameters

Response

If the *reserveAbortConfirmed* operation is successful, a *reserveAbortConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

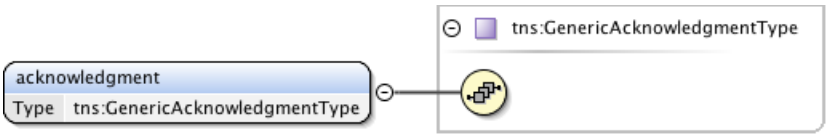

Figure 42 – *reserveAbortConfirmedACK* message structure.

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

8.4.4 *reserveTimeout* message elements

The *reserveTimeout* message is an autonomous message issued from a PA to an RA when a timeout on an existing reserve request occurs, and the PA has freed any uncommitted resources associated with the reservation. This type of event is originated from an uPA managing network resources associated with the reservation, and propagated up the request tree to the originating uRA. An aggregator NSA (performing both a PA and RA role) will map the received *connectionId* into a context understood by its direct parent RA in the request tree, then propagate the event upwards. The originating *connectionId* and uPA are provided in separate elements to maintain the

original context generating the timeout. The *timeoutValue* and *timeStamp* are populated by the originating uPA and propagated up the tree untouched by intermediate NSA.

The *reserveTimeoutACK* indicates that the RA has accepted the *reserveTimeout* event for processing. There is no associated Confirmed or Failed message for this operation.

Type	Direction	Input	Output	Fault
Event	PA to RA	*reserveTimeout*	*reserveTimeoutACK*	*serviceException*

Table 25 *reserveTimeout* message elements

8.4.4.1 Request: *reserveTimeout*

The NSI CS *reserveTimeout* message allows a PA to communicate to the RA a reserve timeout condition on an outstanding reserve operation.

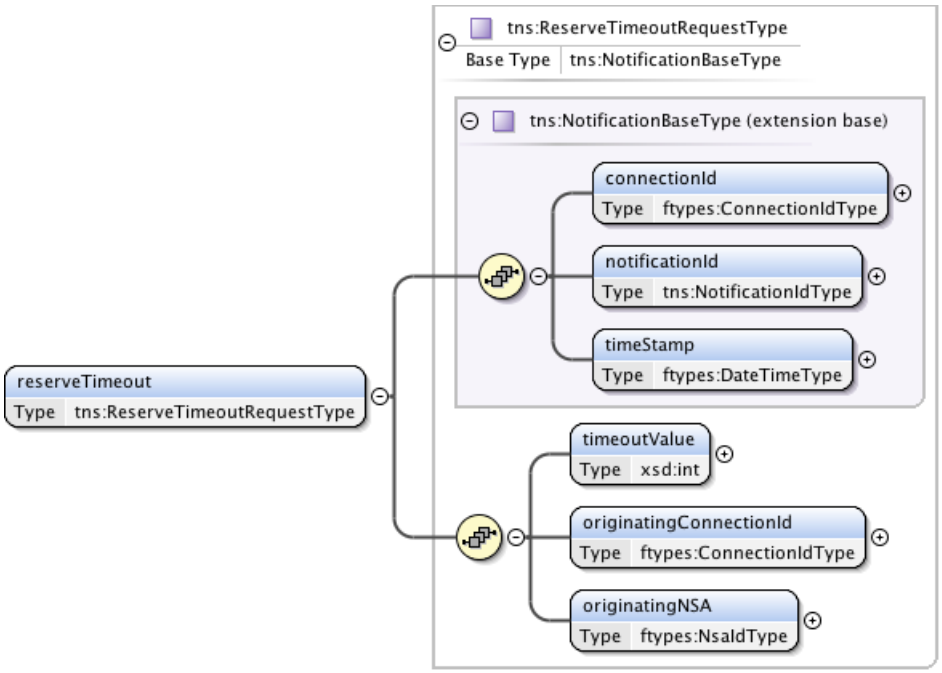

Figure 43 – *reserveTimeout* request message structure.

Parameters

The *reserveTimeout* message has the following parameters:

Parameter	Description
connectionId	The PA assigned *connectionId* that this notification is against.
notificationId	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	Time the event was generated on the originating NSA.
timeoutValue	The timeout value in seconds that expired this reservation.
originatingConnectionId	The *connectionId* that triggered the reserve timeout.
originatingNSA	The NSA originating the timeout event.

Table 26 *reserveTimeout request* parameters

Response

If the *reserveTimeout* operation is successful, a *reserveTimeoutACK* message is returned, otherwise a *serviceException* is returned. An RA sends this *reserveTimeoutACK* message immediately after receiving the *reserveTimeout* event to acknowledge to the PA the *reserveTimeout*

event has been accepted for processing. The *reserveTimeoutACK* message is implemented using the generic acknowledgement message.

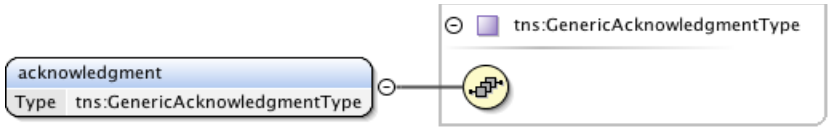

Figure 44 – *reserveTimeoutACK* message structure.

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

8.4.5 *provision* message elements

The *provision* message is sent from an 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
Error	PA to RA	*error*	*errorACK*	*serviceException*

Table 27 *provision* message elements

8.4.5.2 Request: *provision*

The NSI CS *provision* message allows an 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.

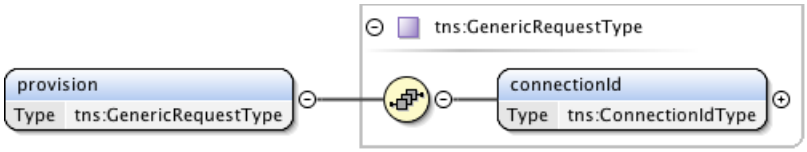

Figure 45 – *provision* request message structure.

Parameters

The *provision* message has the following parameters:

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

Table 28 *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.

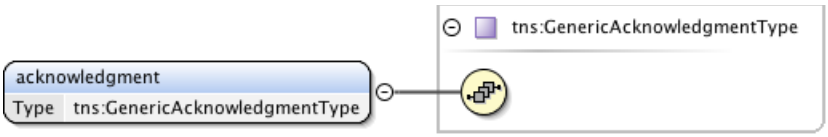

Figure 46 – *provisionACK* message structure.

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

8.4.5.3 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.

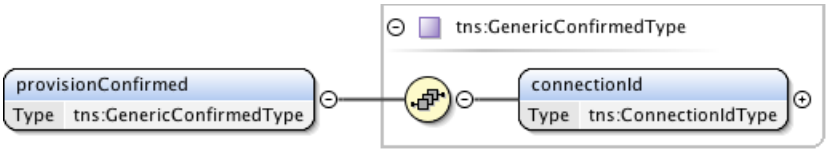

Figure 47 – *provisionConfirmed* message structure.

Parameters

The *provisionConfirmed* message has the following parameters:

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

Table 29 *provisionConfirmed* message parameters

Response

If the *provisionConfirmed* operation is successful, a *provisionConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

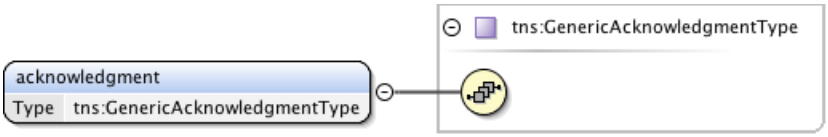

Figure 48 – *provisionConfirmedACK* message structure.

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

8.4.6 *release* message elements

The *release* message is sent from an 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
Error	PA to RA	*error*	*errorACK*	*serviceException*

Table 30 Release message elements

8.4.6.2 Request: *release*

The NSI CS *release* message allows an RA to transition a previously requested reservation into a released state. A reservation in a released state will deactivate associated data plane resources, but the reservation is not affected.

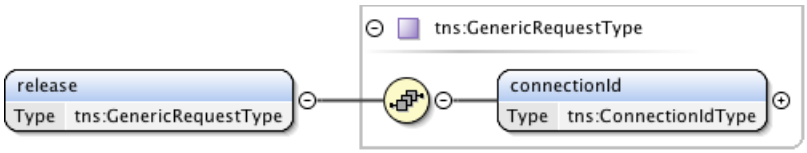

Figure 49 – *release* request message structure.

Parameters

The *release* message has the following parameters:

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

Table 31 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.

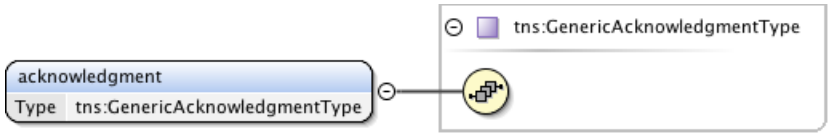

Figure 50 – *releaseACK* message structure.

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

8.4.6.3 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.

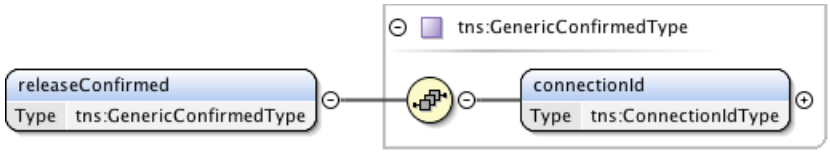

Figure 51 – *releaseConfirmed* message structure.

Parameters

The *releaseConfirmed* message has the following parameters:

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

Table 32 *releaseConfirmed* message parameters

Response

If the *releaseConfirmed* operation is successful, a *releaseConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

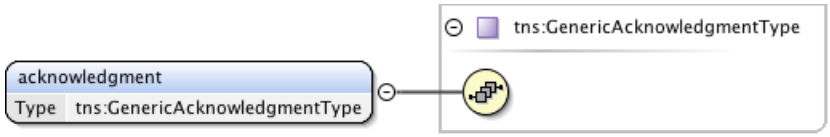

Figure 52 – *releaseConfirmedACK* message structure.

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

8.4.7 *terminate* message elements

The *terminate* message is sent from an 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
Error	PA to RA	error	*errorACK*	*serviceException*

Table 33 *terminate* message elements

8.4.7.1 Request: *terminate*

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

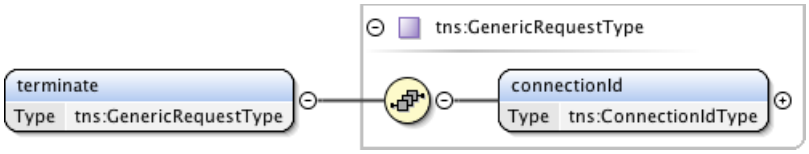

Figure 53 – *terminate* request message structure.

Parameters

The *terminate* message has the following parameters:

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

Table 34 *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.

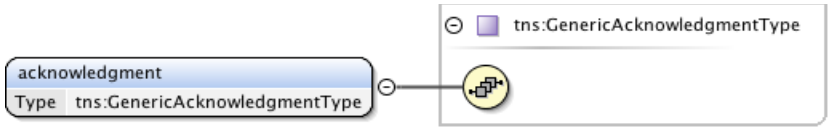

Figure 54 – *terminateACK* message structure.

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

8.4.7.2 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.

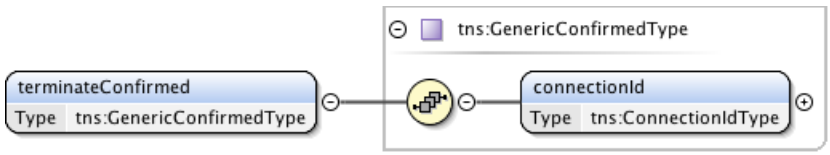

Figure 55 – *terminateConfirmed* message structure.

Parameters

The *terminateConfirmed* message has the following parameters:

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

Table 35 *terminateConfirmed* message parameters

Response

If the *terminateConfirmed* operation is successful, a *terminateConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

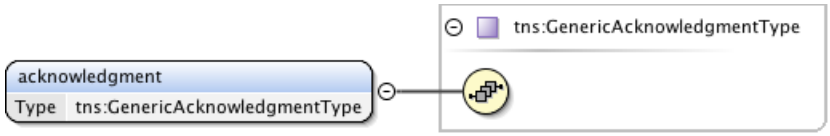

Figure 56 – *terminateConfirmedACK* message structure.

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

8.4.8 *error* message elements

The *error* message is sent from a PA to an RA in response to an outstanding operation request when an error condition encountered, and as a result, the operation cannot be successfully completed. The *correlationId* carried in the NSI CS header structure will identify the original request associated with this error message. The *errorACK* indicates that the RA has accepted the *error* request for processing. There is no associated Confirmed or Failed message for this operation.

Type	Direction	Input	Output	Fault
Request	PA to RA	*error*	*errorACK*	*serviceException*

Table 36 *error* message elements

8.4.8.1 Request: *error*

The NSI CS *error* message allows a PA to communicate to the RA an error condition on an outstanding request operation.

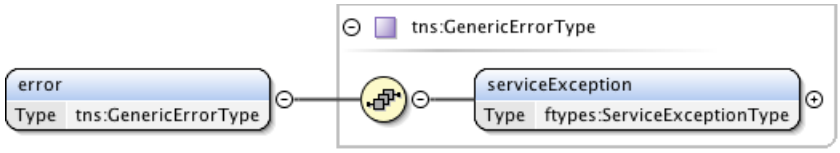

Figure 57 – *error* request message structure.

Parameters

The *error* message has the following parameters:

Parameter	Description
serviceException	Specific error condition and the reason for the failure.

Table 37 *error* message parameters

Response

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

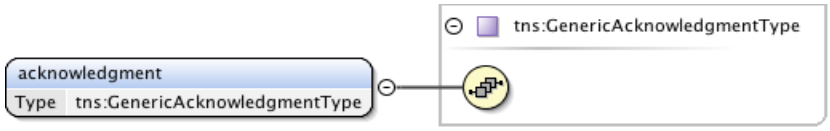

Figure 58 – *errorACK* message structure.

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

8.4.9 *errorEvent* message elements

The *errorEvent* message is an autonomous message issued from a PA to an RA when an existing reservation encounters an autonomous error condition that may impact the reservation. The three errors currently modeled are:

- The reservation is administratively terminated on an uPA before the reservation's scheduled end-time.
- An activation or deactivation of data plane resources associated with the reservation has failed.
- An error has occurred within the data plane that has impacted resources associated with the reservation.

This type of event originates from an uPA managing network resources associated with the reservation, and propagated up the request tree to the originating uRA. An aggregator NSA (performing both a PA and RA role) will map the received *connectionId* into a context understood by its direct parent RA in the request tree, then propagate the event upwards. The originating *connectionId* and uPA are provided in separate elements to maintain the original context generating

the error. The *timeStamp* is populated by the originating uPA and propagated up the tree untouched by intermediate NSA.

The *errorEventACK* indicates that the RA has accepted the *errorEvent* event for processing. There is no associated Confirmed or Failed message for this operation.

Type	Direction	Input	Output	Fault
Event	PA to RA	*errorEvent*	*errorEventACK*	*serviceException*

Table 38 *errorEvent* message elements

8.4.9.1 Request: *errorEvent*

The NSI CS *errorEvent* message allows a PA to communicate to the RA an error condition on an existing reservation.

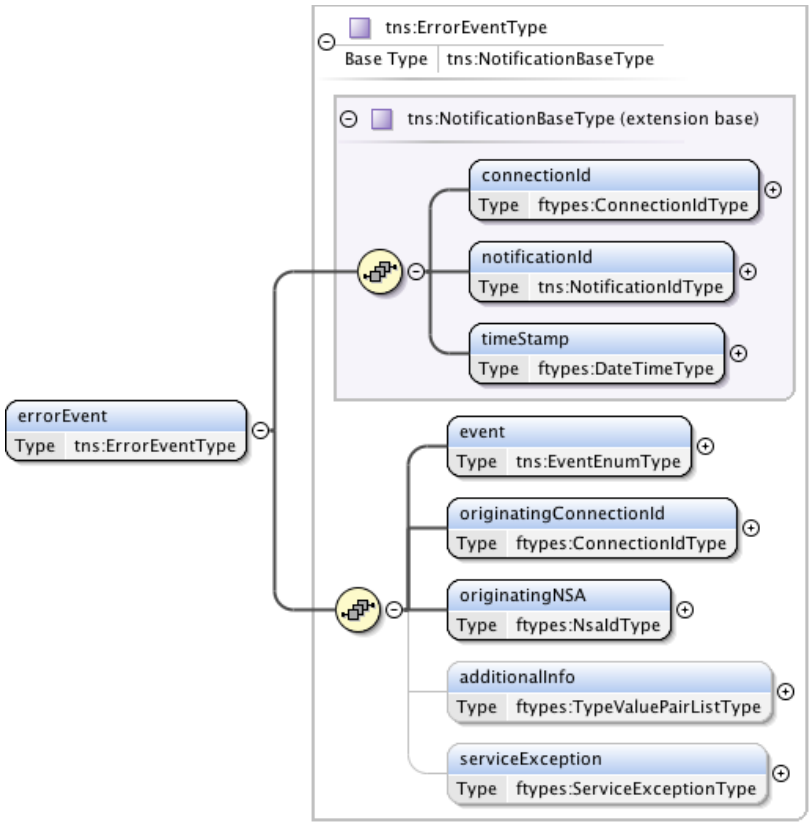

Figure 59 – *errorEvent* request message structure.

Parameters

The *errorEvent* message has the following parameters:

Parameter	Description
connectionId	The PA assigned *connectionId* that this notification is against.
notificationId	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	Time the event was generated on the originating NSA.
event	The type of event that generated this notification.
originatingConnectionId	The *connectionId* that triggered the error event.
originatingNSA	The NSA originating error event.

additionalInfo	Type/value pairs that can provide additional error context as needed.
serviceException	Specific error condition - the reason for the generation of the error event.

Table 39 *reserveTimeout request* parameters

Response

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

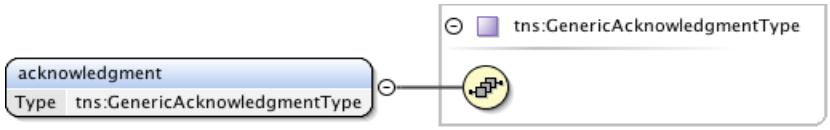

Figure 60 – *errorEventACK* message structure.

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

8.4.10 *dataPlaneStateChange* message elements

The *dataPlaneStateChange* message is an autonomous message issued from a PA to an RA when an existing reservation encounters a data plane state change. Possible data plane status changes are:

- Data plane activation;
- Data plane deactivation;
- Data plane activation version change.

This type of event is originated from an uPA managing network resources associated with the reservation, and propagated up the request tree to the originating uRA. An aggregator NSA (performing both a PA and RA role) will map the received *connectionId* into a context understood by its direct parent RA in the request tree, then propagate the event upwards only if there is a change in the last reported data plane status. The originating *connectionId* and uPA are provided in separate elements to maintain the original context generating the data plane state change. The *timeStamp* is populated by the originating PA and propagated up the tree untouched by intermediate NSA.

The *dataPlaneStateChangeACK* indicates that the RA has accepted the *dataPlaneStateChange* event for processing. There is no associated Confirmed or Failed message for this operation.

Type	Direction	Input	Output	Fault
Event	PA to RA	*dataPlaneStateChange*	*dataPlaneStateChangeACK*	*serviceException*

Table 40 *dataPlaneStateChange* message elements

8.4.10.1 Request: *dataPlaneStateChange*

The NSI CS *dataPlaneStateChange* message allows a PA to communicate to the RA when an existing reservation encounters a data plane state change.

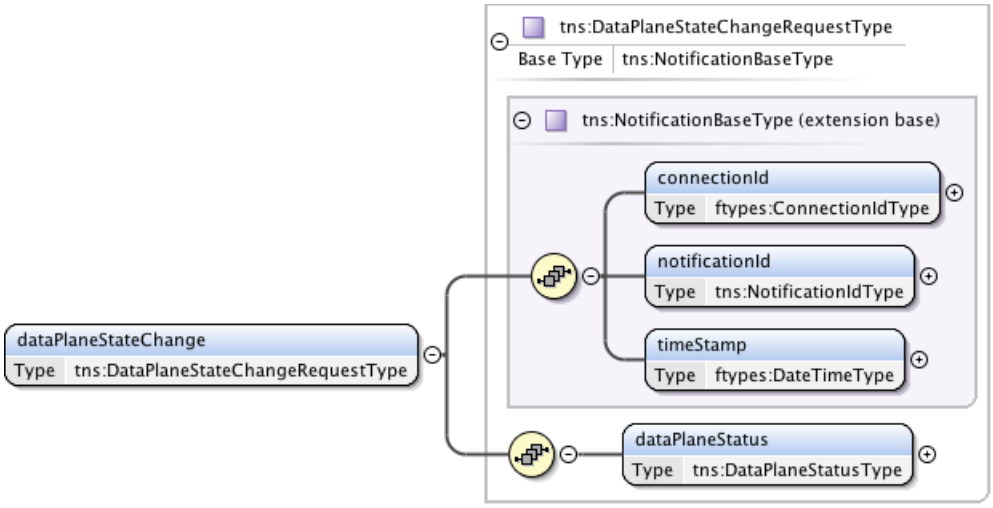

Figure 61 – *dataPlaneStateChange* request message structure.

Parameters

The *dataPlaneStateChange* message has the following parameters:

Parameter	Description
connectionId	The PA assigned *connectionId* that experienced the data plane state change
notificationId	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	Time the event was generated on the originating PA.
dataPlaneStatus	Current data plane activation state for the reservation identified by *connectionId*.

Table 41 *dataPlaneStateChange request* parameters

Response

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

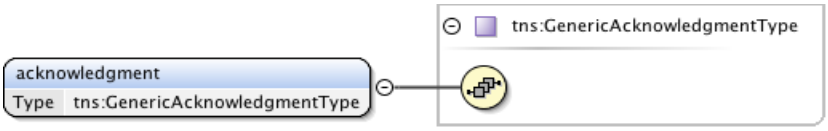

Figure 62 – *dataPlaneStateChangeACK* message structure.

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

8.4.11 *messageDeliveryTimeout* message elements

The *messageDeliveryTimeout* message is an autonomous message issued from a PA to an RA when Message Transport Layer (MTL) delivery or Coordinator timeout has occurred for an outstanding request message within an NSA. This message is issued from the PA that has encountered the error up the request 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 confirmed or failed reply has been received to a previous request issued by the Coordinator. In both cases the timers for these timeout conditions are locally defined.

The *messageDeliveryTimeoutACK* indicates that the RA has accepted the *messageDeliveryTimeout* event for processing. There is no associated Confirmed or Failed message for this operation.

Type	Direction	Input	Output	Fault
Event	PA to RA	*messageDeliveryTimeout*	*messageDeliveryTimeoutACK*	*serviceException*

Table 42 *messageDeliveryTimeout* message elements

8.4.11.1 Request: *messageDeliveryTimeout*

The NSI CS *messageDeliveryTimeout* message allows a PA to communicate to the RA when a Message Transport Layer (MTL) delivery or Coordinator timeout has occurred for an outstanding request message within an NSA.

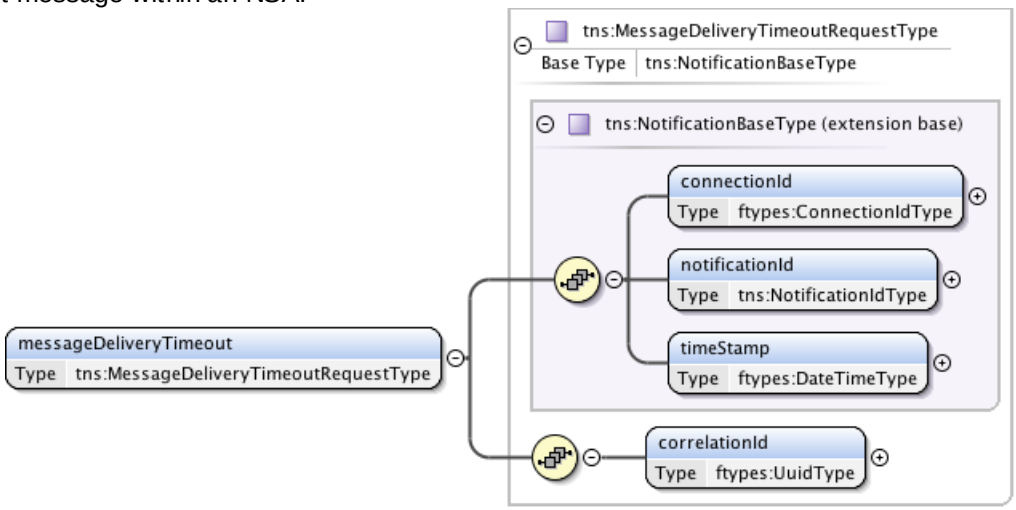

Figure 63 – *messageDeliveryTimeout* request message structure.

Parameters

The *messageDeliveryTimeout* message has the following parameters:

Parameter	Description
connectionId	The PA assigned connectionId that experienced the message delivery timeout
notificationId	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	Time the event was generated on the originating NSA.
correlationId	This value indicates the *correlationId* of the original message that the transport layer failed to send.

Table 43 *messageDeliveryTimeout request* parameters

Response

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

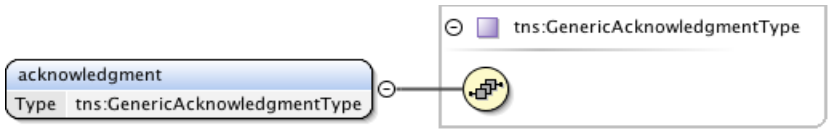

Figure 64 – *messageDeliveryTimeout* message structure.

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

8.4.12 *querySummary* message elements

The *querySummary* message is sent from an 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 *error* 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	N/A	N/A	N/A	N/A
Error	PA to RA	*error*	*errorACK*	*serviceException*

Table 44 *querySummary* message elements

8.4.12.1 Request: *querySummary*

The *querySummary* message provides a mechanism for an 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.

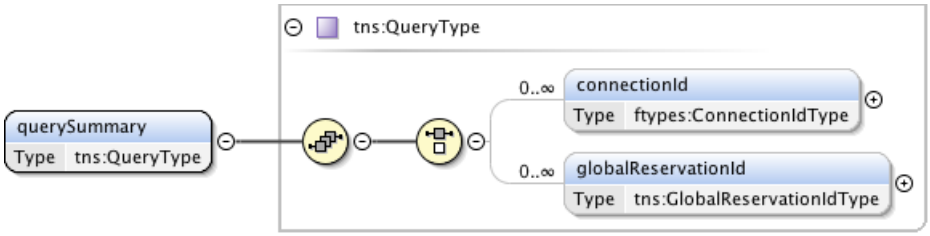

Figure 65 – *querySummary* request message structure.

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 45 *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.

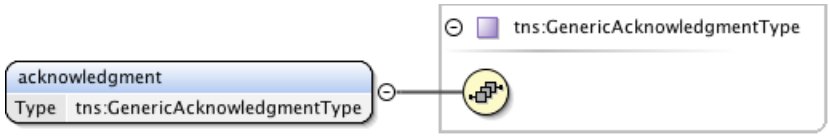

Figure 66 – *querySummaryACK* message structure.

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

8.4.12.2 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.

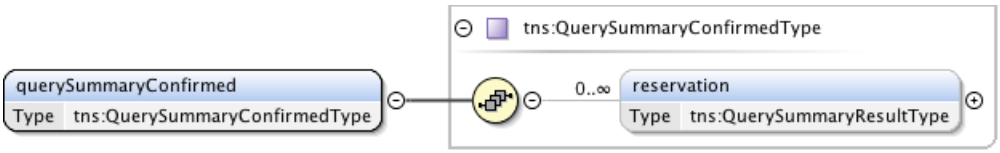

Figure 67 – *querySummaryConfirmed* message structure.

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 46 *querySummaryConfirmed* message parameters

A query will return the currently committed reservation version number, however, if the initial version of the reservation has not yet been committed, the query will return base reservation information (connectionId, globalReservationId, description, requesterNSA, and connectionStates) with no versioned reservation criteria.

Response

If the *querySummaryConfirmed* operation is successful, a *querySummaryConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

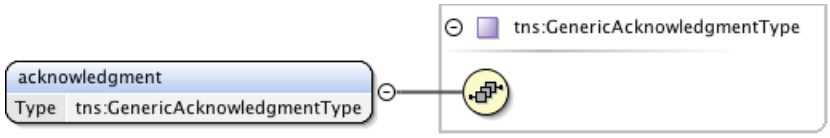

Figure 68 – *querySummaryConfirmedACK* message structure.

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

8.4.12.3 *Error*

An *error* 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 an error message, but instead a *querySummaryConfirmed* with an empty list of reservations. This error message follows that standard error flow defined in section 8.4.8.

8.4.13 *querySummarySync* message elements

The *querySummarySync* message is sent from an 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*	*error*

Table 47 *querySummarySync* message elements

8.4.13.1 Request: *querySummarySync*

The *querySummarySync* message provides a mechanism for an 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.

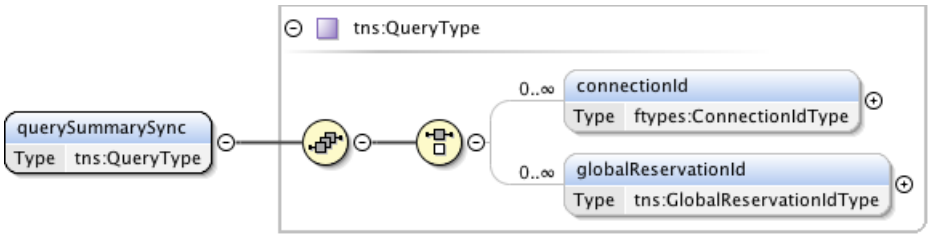

Figure 69 – *querySummarySync* request message structure.

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 48 *querySummarySync* message parameters

Response (Confirmed)

If the *querySummarySync* operation is successful, a *querySummarySyncConfirmed* message is returned directly in the (SOAP) reponse; otherwise a standard *error* message is returned to indicate an error in processing the query has occurred.

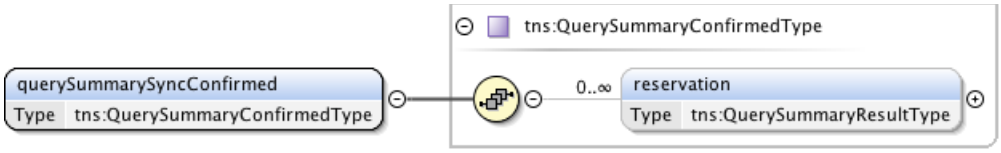

Figure 70 – *querySummarySyncConfirmed* message structure.

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 49 *querySummarySyncConfirmed* message parameters

A query will return the currently committed reservation version number, however, if the initial version of the reservation has not yet been committed, the query will return base reservation information (connectionId, globalReservationId, description, requesterNSA, and connectionStates) with no versioned reservation criteria.

Response (Error)

A standard *error* 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 *error* message, but instead a *querySummarySyncConfirmed* with an empty list. This error message follows that standard error flow defined in section 8.4.8.

8.4.14 *queryRecursive* message elements

The *queryRecursive* message is sent from an 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	N/A	N/A	N/A	N/A
Error	PA to RA	*error*	*errorACK*	*serviceException*

Table 50 *queryRecursive* message elements

8.4.14.1 Request: *queryRecursive*

The *queryRecursive* message provides a mechanism for an 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.

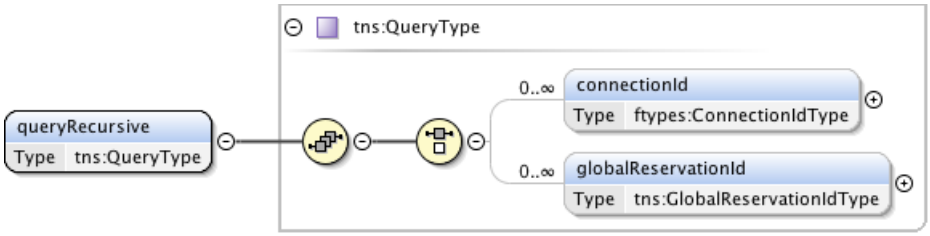

Figure 71 – *queryRecursive* request message structure.

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 51 *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.

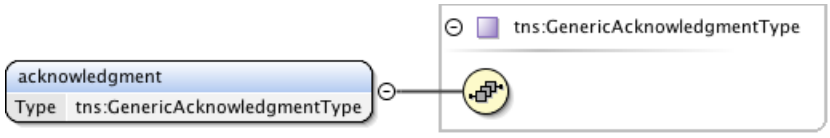

Figure 72 – *queryRecursiveACK* message structure.

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

8.4.14.2 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.

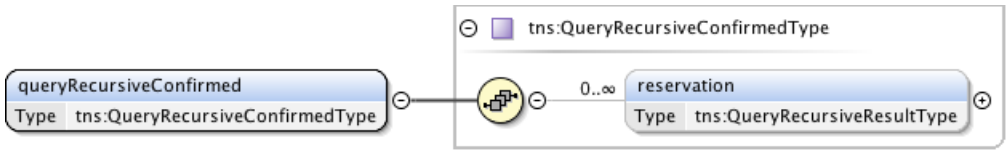

Figure 73 – *queryRecursiveConfirmed* message structure.

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 52 *queryRecursiveConfirmed* message parameters

A query will return the currently committed reservation version number, however, if the initial version of the reservation has not yet been committed, the query will return base reservation information (connectionId, globalReservationId, description, requesterNSA, and connectionStates) with no versioned reservation criteria.

Response

If the *queryRecursiveConfirmed* operation is successful, a *queryRecursiveConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

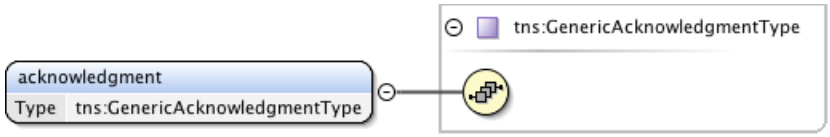

Figure 74 – *queryRecursiveConfirmedACK* message structure.

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

8.4.14.3 Error

An *error* 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 an error message, but instead a *queryRecursive*Confirmed with an empty list of reservations. This error message follows that standard error flow defined in section 8.4.8.

8.4.15 *queryNotification* message elements

The *queryNotification* message is sent from an 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 synchronous version may be used by a polling RA to retrieve the list of notifications messages issued.

The *queryNotificationACK* indicates that the PA has accepted the *queryNotification* request for processing. A *queryNotificationConfirmed* or *generic error* 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	N/A	N/A	N/A	N/A
Error	PA to RA	*error*	*errorACK*	*serviceException*

Table 53 *queryNotification* message elements

8.4.15.1 Request: *queryNotification*

The *queryNotification* message provides a mechanism for an 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*.

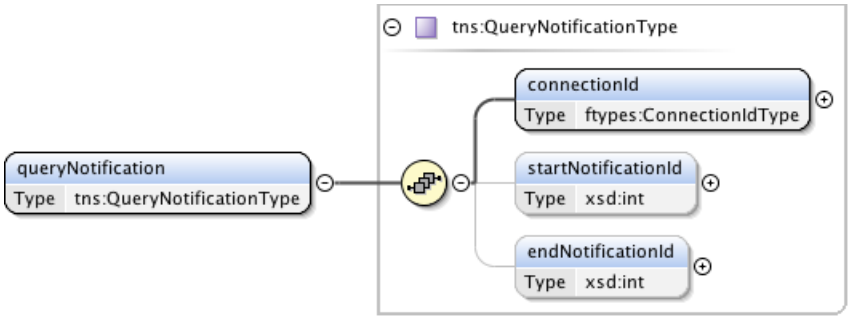

Figure 75 – *queryNotification* request message structure.

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 54 *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.

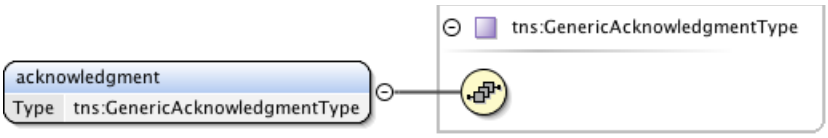

Figure 76 – *queryNotificationACK* message structure.

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

8.4.15.2 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.

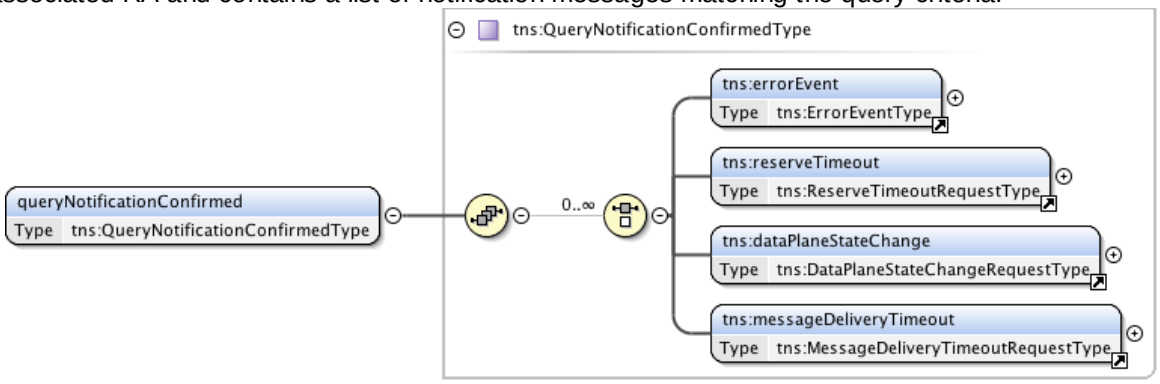

Figure 77 – *queryNotificationConfirmed* message structure.

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.
dataPlaneStateChange	A data plane state change notification.
messageDeliveryTimeout	A set of zero or more message delivery timeout notification.

Table 55 *queryNotificationConfirmed* message parameters

Response

If the *queryNotificationConfirmed* operation is successful, a *queryNotificationConfirmedACK* message is returned, otherwise a *serviceException* is returned. An 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.

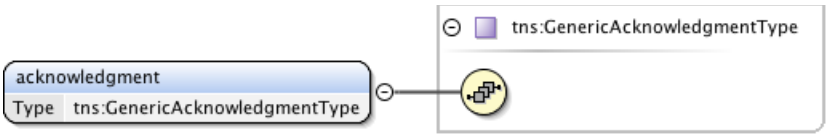

Figure 78 – *queryNotificationConfirmedACK* message structure.

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

8.4.15.3 *Error*

An *error* 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 reservations does not result in an error message, but instead a *queryNotification*Confirmed with an empty list of reservations. This error message follows that standard error flow defined in section 8.4.8.

8.4.16 *queryNotificationSync* message elements

The *queryNotificationSync* message is sent from an 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 standard *error* 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*	*error*

Table 56 *queryNotificationSync* message elements

8.4.16.1 Request: *queryNotificationSync*

The *queryNotificationSync* message provides a mechanism for an 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*.

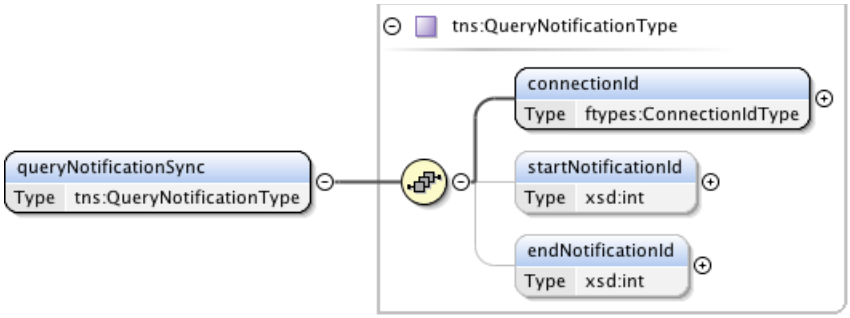

Figure 79 – *queryNotificationSync* request message structure.

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 57 *queryNotificationSync* message parameters

Response (Confirmed)

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

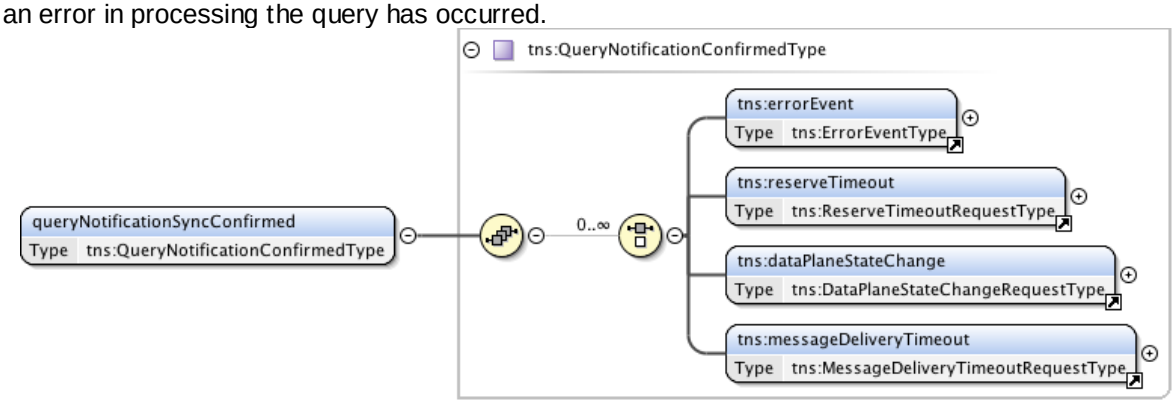

Figure 80 – *queryNotificationSyncConfirmed* message structure.

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.
dataPlaneStateChange	A data plane state change notification.
messageDeliveryTimeout	A set of zero or more message delivery timeout notification.

Table 58 *queryNotificationSyncConfirmed* message parameters

Response (Error)

A standard *error* message structure 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 *error* message, but instead a *queryNotificationSyncConfirmed* with an empty list.

8.4.17 *queryResult* message elements

The *queryResult* message is sent from an RA to a PA to retrieve operation result messages (confirmed, failed, and error) against an existing reservation residing on the PA. The returned results will be a list of confirmed, failed, and error messages for the specified *connectionId*.

The *queryResultACK* indicates that the PA has accepted the *queryResult* request for processing. A *queryResultConfirmed* or generic *error* message will be sent asynchronously to the RA when *queryResult* processing has completed.

Type	Direction	Input	Output	Fault
Request	RA to PA	*queryResult*	*queryResultACK*	*serviceException*
Confirmed	PA to RA	*queryResultConfirmed*	*queryResultConfirmedACK*	*serviceException*
Failed	N/A	N/A	N/A	N/A
Error	PA to RA	*error*	*errorACK*	*serviceException*

Table 59 *queryResult* message elements

8.4.17.1 Request: *queryResult*

The *queryResult* message provides a mechanism for an RA to query the PA for a list of operation result messages (confirmed, failed, and error) against a *connectionId*. An RA can recover lost result messages using this operation.

Elements compose a filter for specifying the results to return in response to the query operation. The filter query provides an inclusive range of result identifiers based on *connectionId*. The result identifier is a sequentially increasing value maintained by the PA for each confirmed, failed, or error message generated by the PA in the context of a single *connectionId*. This identifier is not returned in the individual confirmed, failed, or error messages as with notification, however, it is tracked against a reservation and returned in the reservation query for utilization by polling clients.

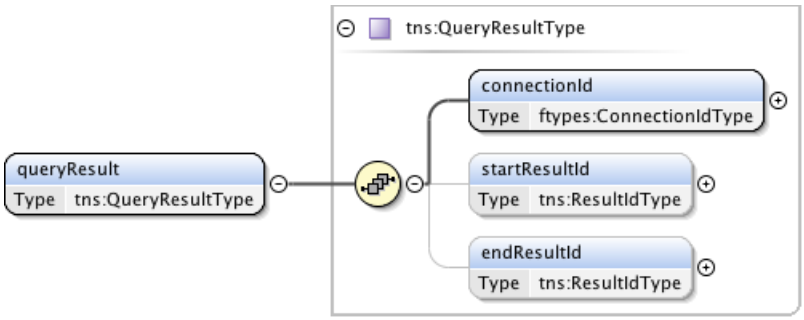

Figure 81 – *queryResult* request message structure.

Parameters

The *queryResult* message has the following parameters:

Parameter	Description
connectionId	Return results for this *connectionId*.
startResultId	The start of the range of *resultId*s to return. If not present then the query should start from oldest *resultId* available.
endResultId	The end of the range of *resultId*s to return. If not present then the query should end with the newest *resultId* available.

Table 60 *queryResult* message parameters

Response

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

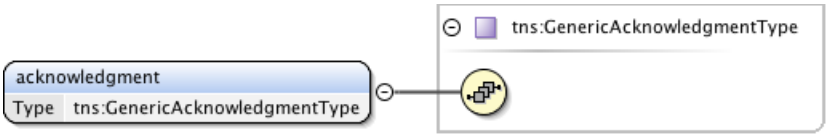

Figure 82 – *queryResultACK* message structure.

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

8.4.17.2 Confirmation: *queryResultConfirmed*

This *queryResultConfirmed* message is sent from the PA to RA as an indication of a successful *queryResult* operation. This is in response to an original *queryResult* request from the associated RA and contains a list of confirmed, failed, and error messages matching the query criteria.

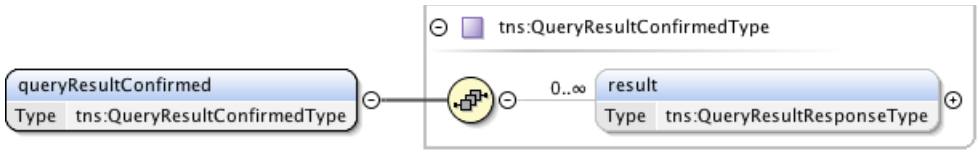

Figure 83 – *queryResultConfirmed* message structure.

Parameters

The *queryResultConfirmed* message has the following parameters:

Parameter	Description
result	Zero or more result elements based on the results matching the specified query.

Table 61 *queryResultConfirmed* message parameters.

Each result returned in the *queryResultConfirmed* message structure will containing a single operation result of the type *QueryResultResponseType* as shown in **Figure 84**.

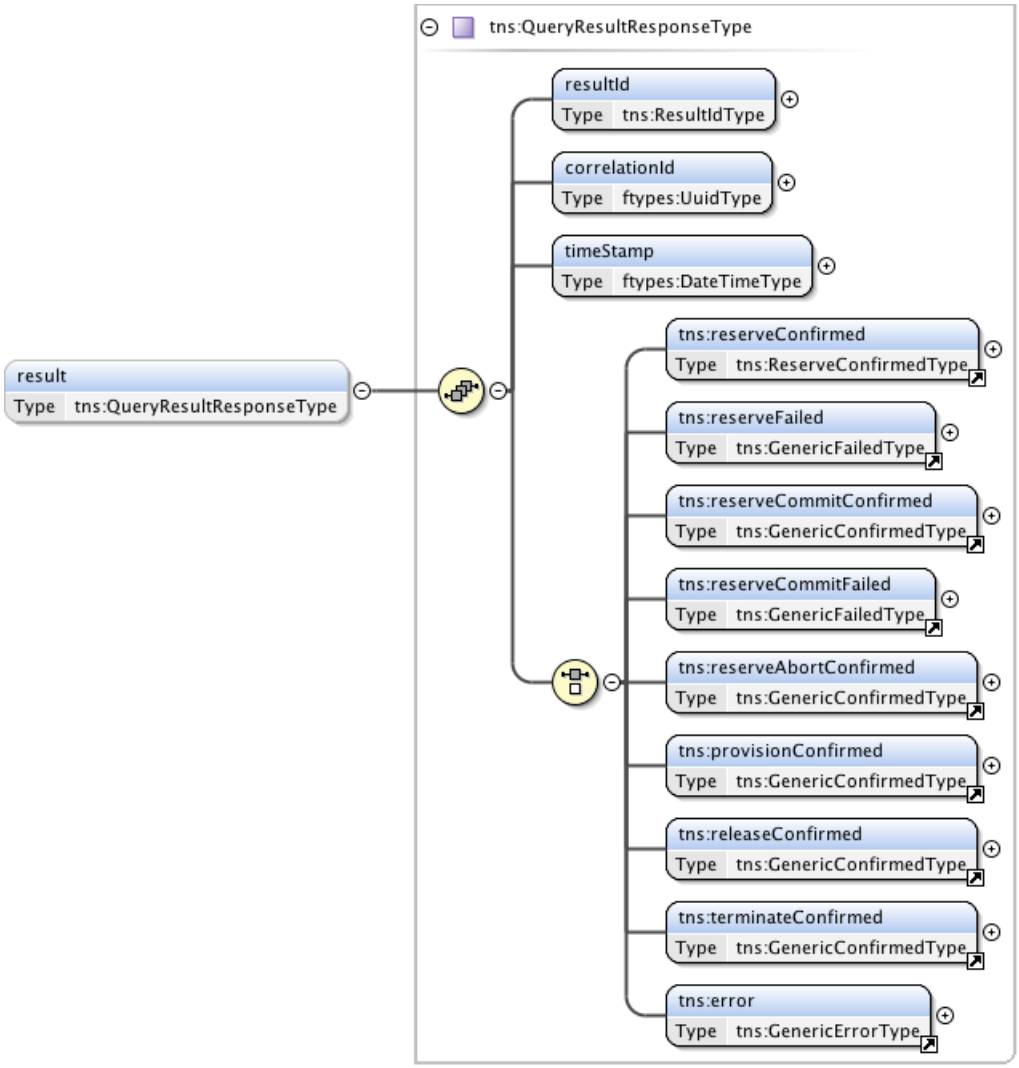

Figure 84 – *QueryResultResponseType* structure.

Parameters

The *QueryResultResponseType* message has the following parameters:

Parameter	Description
resultId	A result identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for sequencing results in the order in which they were generated in the context of the *connectionId*.
correlationId	The *correlationId* corresponding to the operation result as would have been returned in the NSI header element when this result was returned to the RA.
timeStamp	The time this result was generated.
Choice of:	
reserveConfirmed	*Reserve operation confirmation.*
reserveFailed	*Reserve operation failure.*
reserveCommitConfirmed	*Reserve commit operation confirmation.*
reserveCommitFailed	*Reserve commit operation failure.*
reserveAbortConfirmed	*Reserve abort operation confirmation.*
provisionConfirmed	*Provision operation confirmation.*
releaseConfirmed	*Release operation confirmation.*
terminateConfirmed	*Terminate confirmation.*
error	*Error response message.*

Table 62 *QueryResultResponseType* message parameters

Response

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

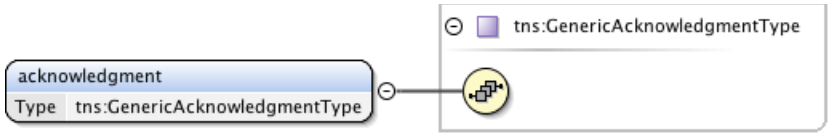

Figure 85 – *queryResultConfirmedACK* message structure.

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

8.4.17.3 Error

An *error* message is sent from the PA to RA as an indication of a *queryResult* operation failure. This is in response to an original *queryResult* request from the associated RA. It is important to note that a *queryResult* operation that results in no matching reservations does not result in an error message, but instead a *queryResult*Confirmed with an empty list of reservations. This error message follows that standard error flow defined in section 8.4.8.

8.4.18 *queryResultSync* message elements

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

Type	Direction	Input	Output	Fault
Request	RA to PA	*queryResultSync*	*queryResultSyncConfirmed*	*error*

Table 63 *queryResultSync* message elements

8.4.18.1 Request: *queryResultSync*

The *queryResultSync* message provides a mechanism for an RA to query the PA for a list of confirmed, failed, and error messages against a *connectionId*. An RA can recover lost result messages using this operation, or a polling RA can use it to retrieve a list of result messages for operations issued.

Elements compose a filter for specifying the results to return in response to the query operation. The filter query provides an inclusive range of result identifiers based on *connectionId*.

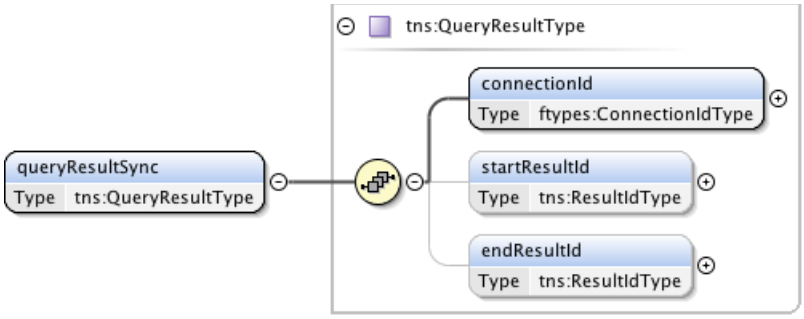

Figure 86 – *queryResultSync* request message structure.

Parameters

The *queryResultSync* message has the following parameters:

Parameter	Description
connectionId	Return results for this *connectionId*.
startResultId	The start of the range of *resultId*s to return. If not present then the query should start from oldest *resultId* available.
endResultId	The end of the range of *resultId*s to return. If not present then the query should end with the newest *resultId* available.

Table 64 *queryResultSync* message parameters.

Response (Confirmed)

If the *queryResultSync* operation is successful, a *queryResultSyncConfirmed* message is returned; otherwise a standard *error* message is returned to indicate an error in processing the query has occurred.

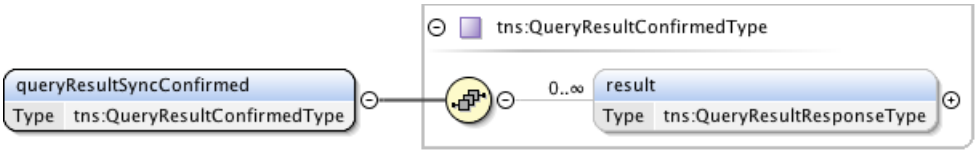

Figure 87 – *queryResultSyncConfirmed* message structure.

Parameters

The *queryResultConfirmed* message has the following parameters:

Parameter	Description
result	Zero or more result elements based on the results matching the specified query.

Table 65 *queryResultSyncConfirmed* message parameters.

Each result returned in the *queryResultSyncConfirmed* message structure will containing a single operation result of the type *QueryResultResponseType* as shown in Figure 84.

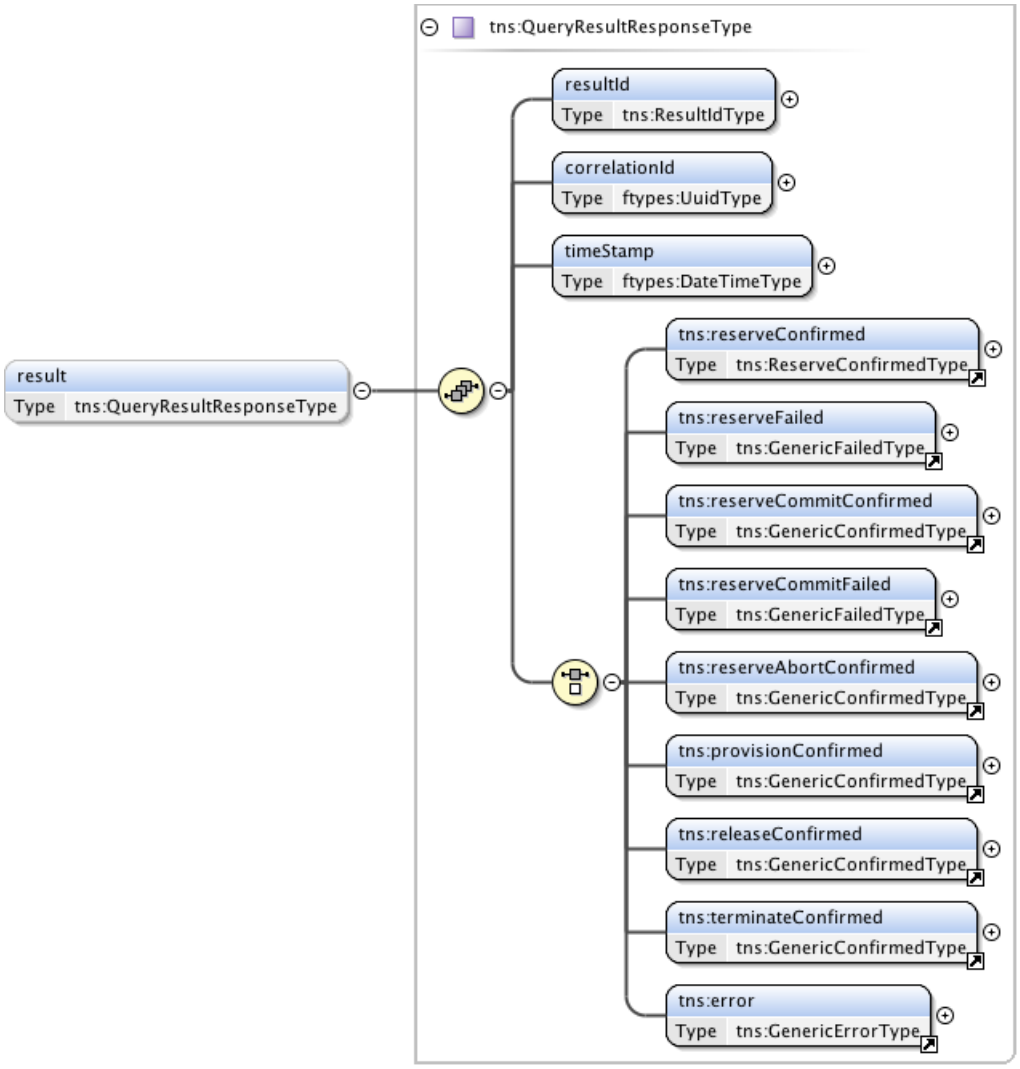

Figure 88 – *QueryResultResponseType* structure.

Parameters

The *queryNotificationConfirmed* message has the following parameters:

Parameter	Description
resultId	A result identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for sequencing results in the order in which they were generated in the context of the *connectionId*.
correlationId	The *correlationId* corresponding to the operation result as would have been returned in the NSI header element when this result was returned to the RA.
timeStamp	The time this result was generated.

Choice of:	
reserveConfirmed	*Reserve operation confirmation.*
reserveFailed	*Reserve operation failure.*
reserveCommitConfirmed	*Reserve commit operation confirmation.*
reserveCommitFailed	*Reserve commit operation failure.*
reserveAbortConfirmed	*Reserve abort operation confirmation.*
provisionConfirmed	*Provision operation confirmation.*
releaseConfirmed	*Release operation confirmation.*
terminateConfirmed	*Terminate confirmation.*
error	*Error response message.*

Table 66 *queryResultConfirmed* message parameters

Response (Error)

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

8.5 NSI CS specific types

Namespace definition: http://schemas.ogf.org/nsi/2013/12/connection/types

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

8.5.1 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.

8.5.1.1 *ChildRecursiveListType*

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

Figure 89 – *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 67 *ChildRecursiveListType* message parameters

8.5.1.2 *ChildRecursiveType*

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 child NSA in either a tree and chain configuration.

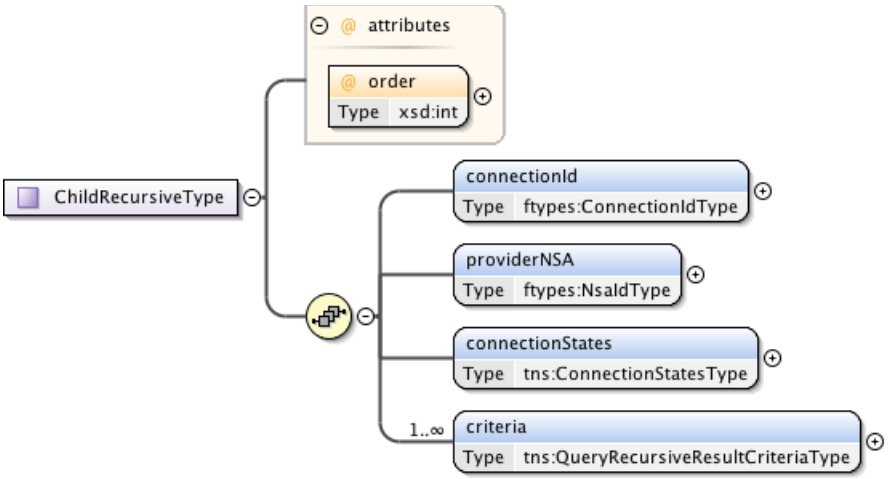

Figure 90 – *ChildRecursiveType*.

Parameters

The *ChildRecursiveType* 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 provider NSA 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 68 *ChildRecursiveType* message parameters.

8.5.1.3 *ChildSummaryListType*

A holder element containing a list of child NSA and their associated connection information. Utilized by the *QuerySummaryResultCriteriaType* to provide a nested list structure of summary reservation information.

Figure 91 – *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 69 *ChildSummaryListType* message parameters.

8.5.1.4 *ChildSummaryType*

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.

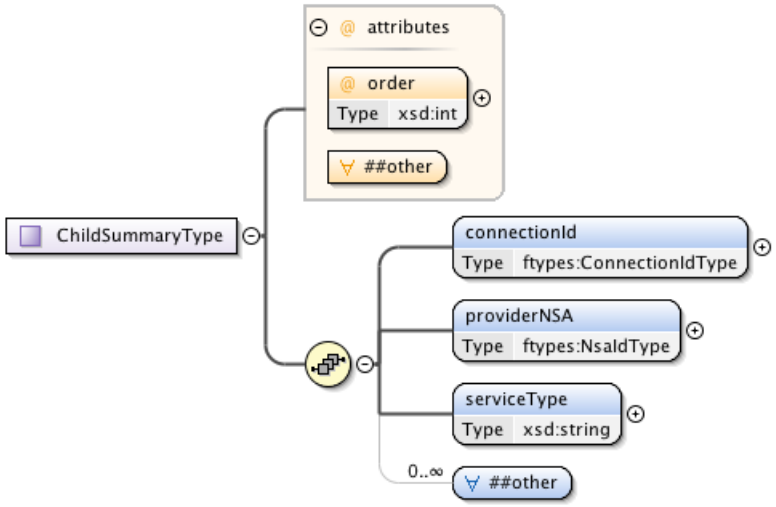

Figure 92 – *ChildSummaryType*.

Parameters

The *ChildSummaryType* 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 provider NSA holding the connection information associated with this instance of data.
serviceType	M	The specific service type of this reservation. This service type string maps into the list of supported service definitions defined by the network providers. In turn, the service type specifies the specific service elements carried in an instance of this type (through the ANY definition) that is associated with the requested service. This element is mandatory.
##other	O	Provides a flexible mechanism allowing additional elements to be provided such as the service specific attributes specified by *serviceType*. Additional 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 70 *ChildSummaryType* message parameters.

8.5.1.5 *ConnectionStatesType*

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

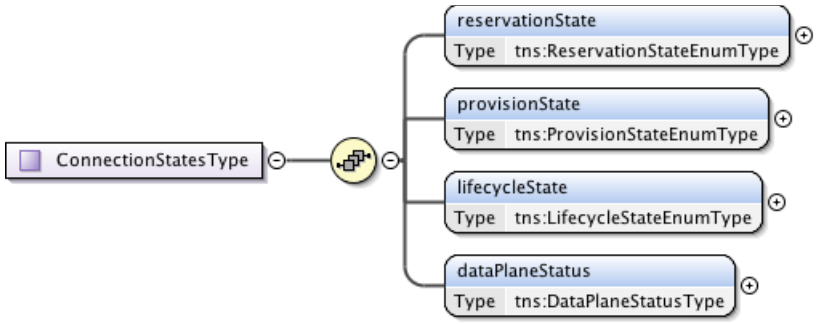

Figure 93 – *ConnectionStatesType*.

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.
lifecycleState	M	Models the current connection lifecycle state.
dataPlaneStatus	M	Models the current connection data plane activation state.

Table 71 *ConnectionStatesType* message parameters

8.5.1.6 *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.

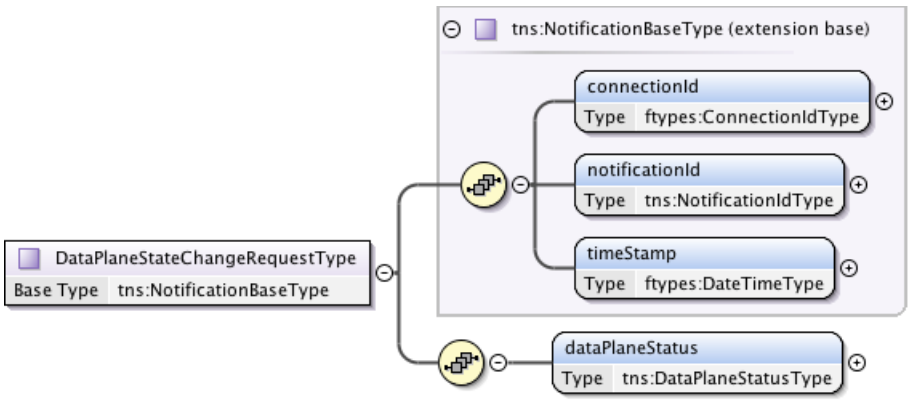

Figure 94 – *DataPlaneStateChangeRequestType*.

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 72 *DataPlaneStateChangeRequestType* message parameters

8.5.1.7 *DataPlaneStatusType*

Models the current connection activation state within the data plane.

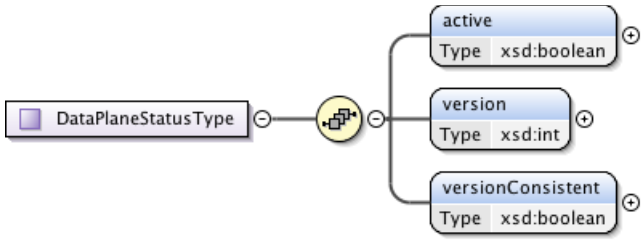

Figure 95 – *DataPlaneStatusType*.

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 73 *DataPlaneStatusType* message parameters

8.5.1.8 *ErrorEventType*

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

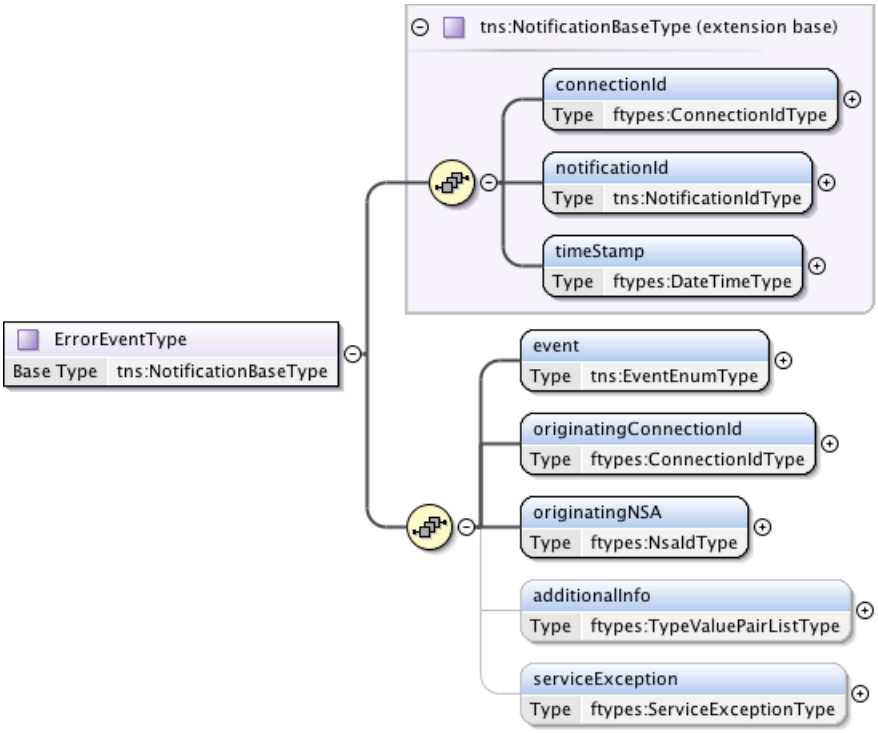

Figure 96 – *ErrorEventType*.

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.
originatingConnectionId	M	The connectionId that triggered the error event.
originatingNSA	M	The NSA originating the error event.
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 74 *ErrorEventType* message parameters

8.5.1.9 *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 97 – *GenericAcknowledgmentType*.

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

8.5.1.10 *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*.

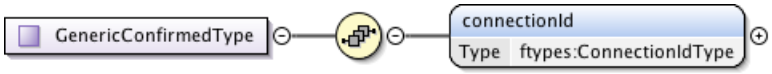

Figure 98 – *GenericConfirmedType*.

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 75 *GenericConfirmedType* message parameters

8.5.1.11 *GenericErrorType*

A generic "Error" message type sent in response to a previous protocol "Request" message. An error message is generated when an error condition occurs that does not result in a state machine transition. This type is used in response to all request types that can return an error.

The *correlationId* carried in the NSI header will identify the original request associated with this error message.

Figure 99 – *GenericErrorType*.

Parameters

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

Parameter	M/O	Description
serviceException	M	Specific error condition indicating the reason for the failure.

Table 76 *GenericErrorType* message parameters

8.5.1.12 *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.

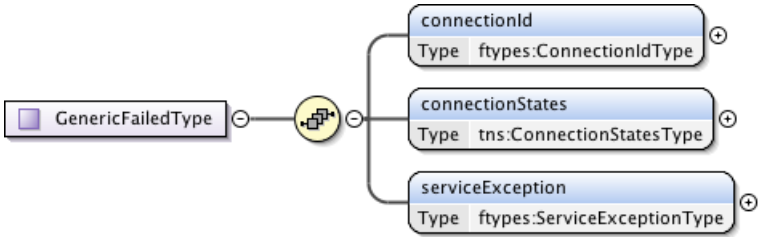

Figure 100 – *GenericFailedType*.

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 77 *GenericFailedType* message parameters

8.5.1.13 *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.

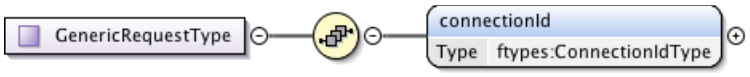

Figure 101 – *GenericRequestType*.

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 78 *GenericRequestType* message parameters

8.5.1.14 *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 confirmed or failed 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.

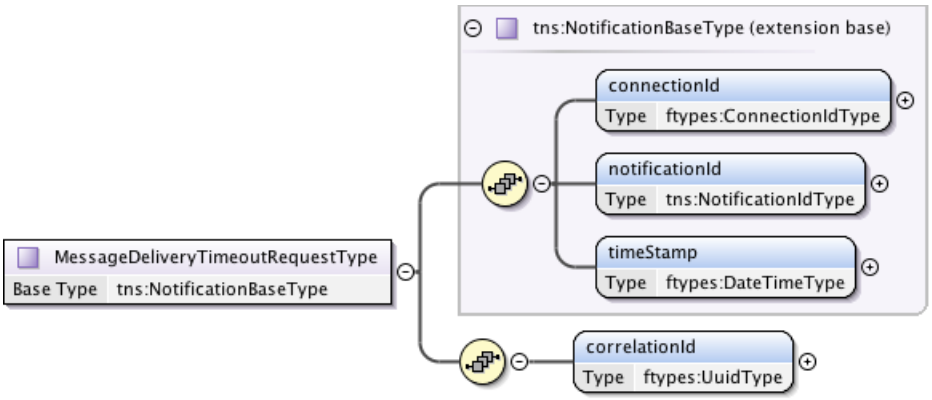

Figure 102 – *MessageDeliveryTimeoutRequestType*.

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 79 *MessageDeliveryTimeoutRequestType* message parameters.

8.5.1.15 *NotificationBaseType*

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

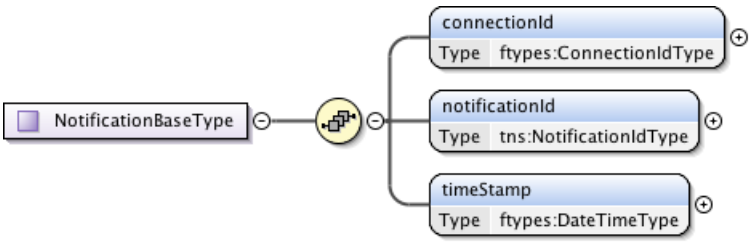

Figure 103 – *NotificationBaseType*.

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 80 *NotificationBaseType* message parameters.

8.5.1.16 *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.

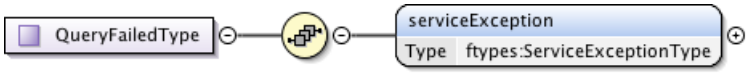

Figure 104 – *QueryFailedType*.

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 81 *QueryFailedType* message parameters

8.5.1.17 *QueryNotificationConfirmedType*

A query notification confirmation containing a list of notification messages matching the specified query criteria.

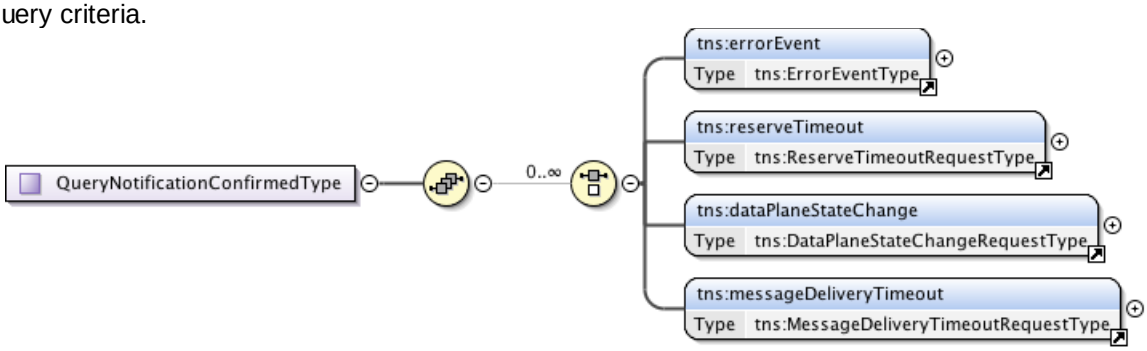

Figure 105 – *QueryNotificationConfirmedType*.

Parameters

The *QueryNotificationConfirmedType* is an optional choice of zero or more of the following parameters (M = Mandatory, O = Optional):

Parameter	M/O	Description
errorEvent	O	Specific error condition - the reason for the failure.
reserveTimeout	O	Reserve timeout notification.
dataPlaneStateChange	O	A data plane state change notification.
messageDeliveryTimeout	O	Message delivery timeout notification.

Table 82 *QueryNotificationConfirmedType* message parameters

8.5.1.18 *QueryNotificationType*

Type definition for the *QueryNotification* message providing a mechanism for a Requester NSA to query a Provider NSA for a set of notifications against a specific *connectionId*.

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.

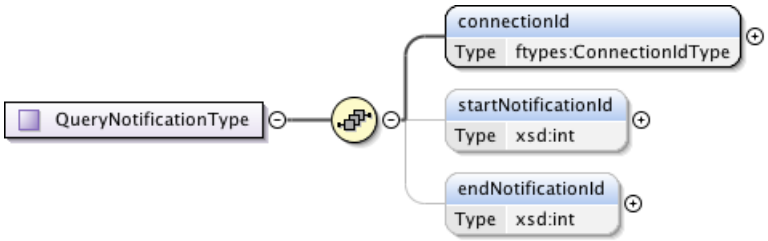

Figure 106 – *QueryNotificationType*.

Parameters

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

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

Table 83 *QueryNotificationType* message parameters

8.5.1.19 *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.

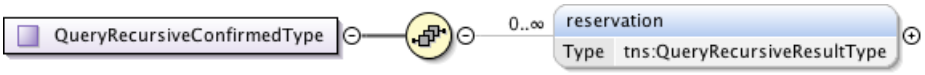

Figure 107 – *QueryRecursiveConfirmed*Type.

Parameters

The *QueryRecursiveConfirmed*Type has 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 84 *QueryRecursiveConfirmed*Type message parameters

8.5.1.20 *QueryRecursiveResultCriteriaType*

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

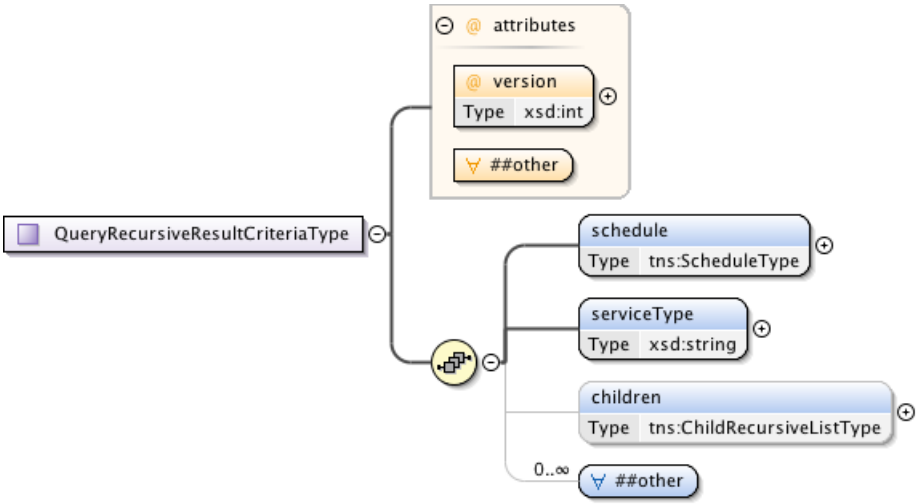

Figure 108 – *QueryRecursiveResultCriteriaType*.

Parameters

The *QueryRecursiveResultCriteriaType* 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.
serviceType	M	The specific service type of this reservation. This service type string maps into the list of supported service definitions defined by the network providers. In turn, the service type specifies the specific service elements carried in an instance of this type (through the ANY definition) associated with the requested service.
children	O	If this connection reservation is aggregating child connections then this element contains detailed information about the child connection segment. The level of detail include is left up to the individual NSA and their authorization policies.
any ##other	O	Provides a flexible mechanism allowing additional elements to be provided such as the service-specific parameters specified by *serviceType*. Additional 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 85 *QueryRecursiveResultCriteriaType* message parameters

8.5.1.21 *QueryRecursiveResultType*

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

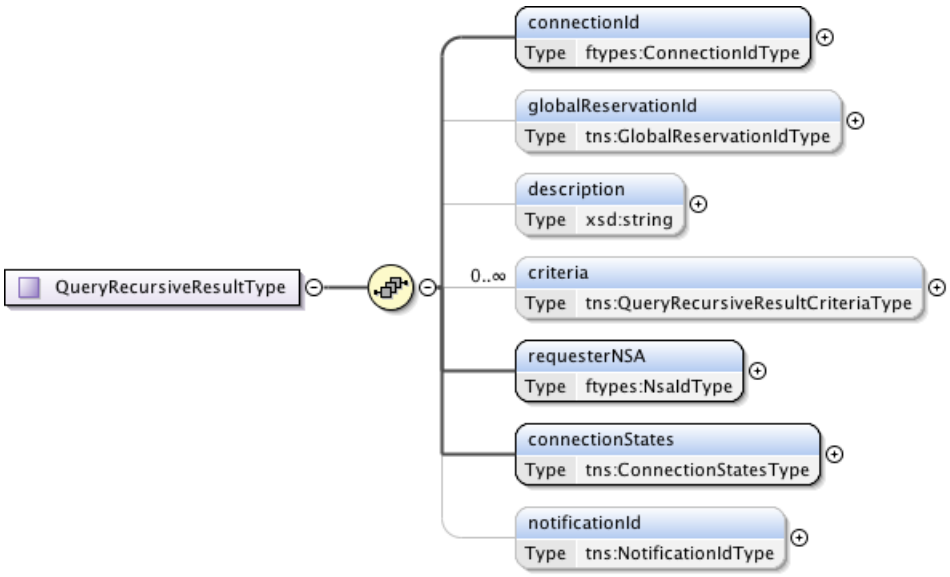

Figure 109 – *QueryRecursiveResultType*.

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 86 *QueryRecursiveResultType* message parameters.

8.5.1.22 *QueryResultConfirmedType*

Type definition for the QueryResultConfirmedType providing a mechanism for a Requester NSA to get a list of Confirmed, Failed, or Error results against a specific *connectionId*.

Figure 110 – *QueryResultConfirmedType*.

Parameters

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

Parameter	M/O	Description
result	O	Zero or more result elements based on the results matching the specified query.

Table 87 *QueryResultConfirmedType* message parameters.

8.5.1.23 QueryResultResponseType

A QueryResultResponseType type containing a single operation result matching the specified query criteria.

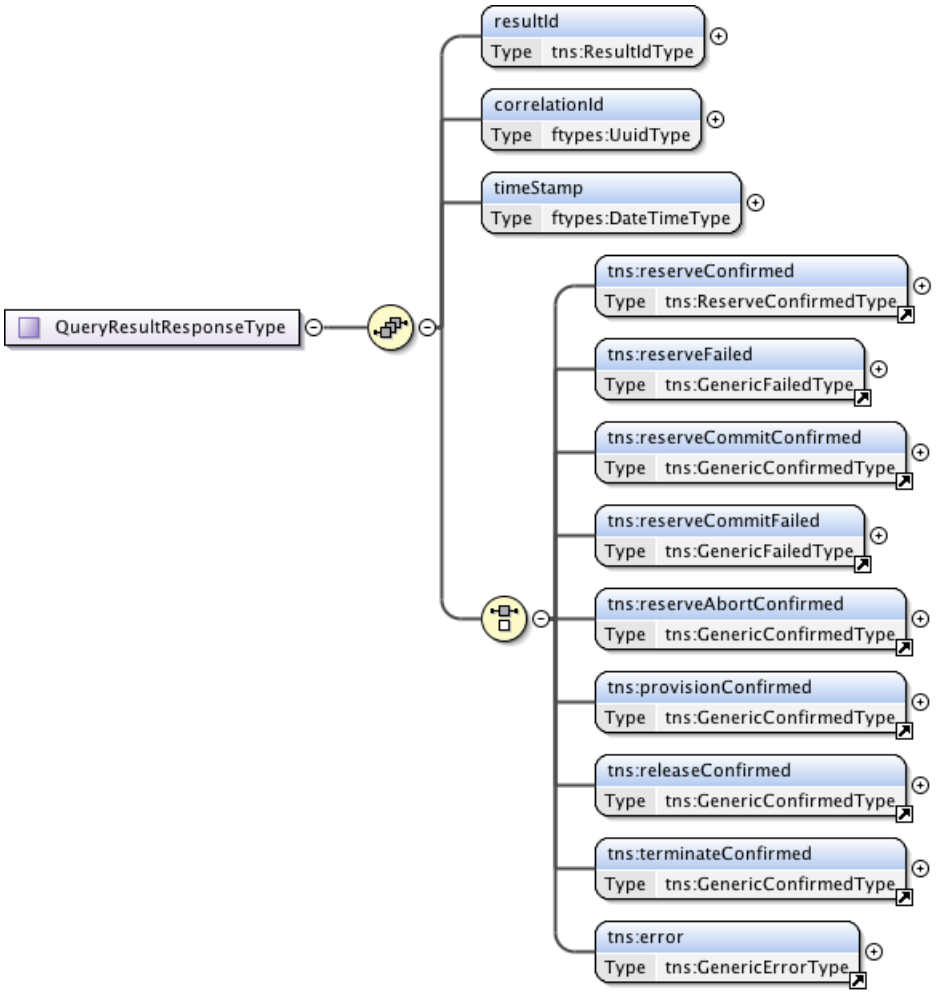

Figure 111 – *QueryResultResponseType* structure.

Parameters

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

Parameter	M/O	Description
resultId	M	A result identifier that is unique in the context of a *connectionId*. This is a linearly increasing identifier that can be used for sequencing results in the order in which they were generated in the context of the *connectionId*.
correlationId	M	The *correlationId* corresponding to the operation result as would have been returned in the NSI header element when this result was returned to the RA.
timeStamp	M	The time this result was generated.
Choice of: *reserveConfirmed* *reserveFailed* *reserveCommitConfirmed* *reserveCommitFailed*	*M*	 *Reserve operation confirmation.* *Reserve operation failure.* *Reserve commit operation confirmation.* *Reserve commit operation failure.*

reserveAbortConfirmed		*Reserve abort operation confirmation.*
provisionConfirmed		*Provision operation confirmation.*
releaseConfirmed		*Release operation confirmation.*
terminateConfirmed		*Terminate confirmation.*
error		*Error response message.*

Table 88 *QueryResultResponseType* message parameters

8.5.1.24 *QueryResultType*

The queryResultType message provides a mechanism for a Requester NSA to query a Provider NSA for a set of Confirmed, Failed, or Errors results against a specific *connectionId*.

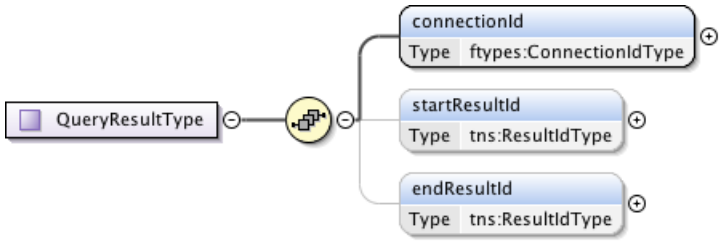

Figure 112 – *QueryResult*Type.

Parameters

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

Parameter	M/O	Description
connectionId	M	Retrieve results for this *connectionId*.
startResultId	O	The start of the range of result Ids to return. If not present, then the query should start from oldest result available.
endResultId	O	The end of the range of result Ids to return. If not present then the query should end with the newest result available.

Table 89 *QueryResult*Type message parameters.

8.5.1.25 *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.

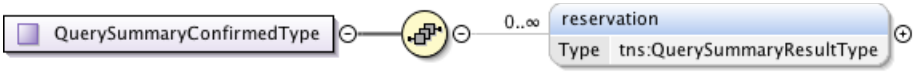

Figure 113 – *QuerySummaryConfirmed*Type.

Parameters

The *QuerySummaryConfirmed*Type has 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 90 *QuerySummaryConfirmed*Type message parameters.

8.5.1.26 *QuerySummaryResultCriteriaType*

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

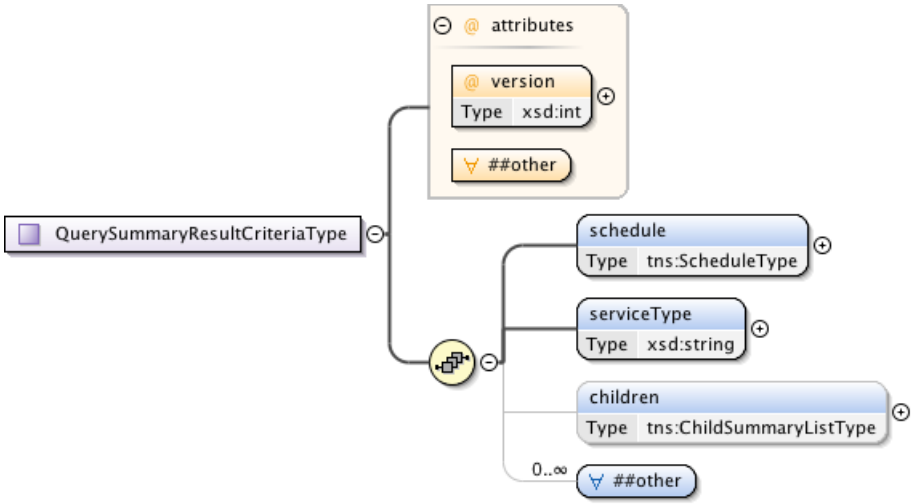

Figure 114 – *QuerySummaryResultCriteriaType*.

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.
serviceType	M	The specific service type of this reservation. This service type string maps into the list of supported service definitions defined by the network providers. In turn, the service type specifies the specific service elements carried in an instance of this type (through the ANY definition) associated with the requested service.
children	O	If this connection reservation is aggregating child connections then this element contains summary information about the child connection segment.
any ##other	O	Provides a flexible mechanism allowing additional elements to be provided such as the service-specific parameters specified by *serviceType*. Additional 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 91 *QuerySummaryResultCriteriaType* message parameters.

8.5.1.27 *QuerySummaryResultType*

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

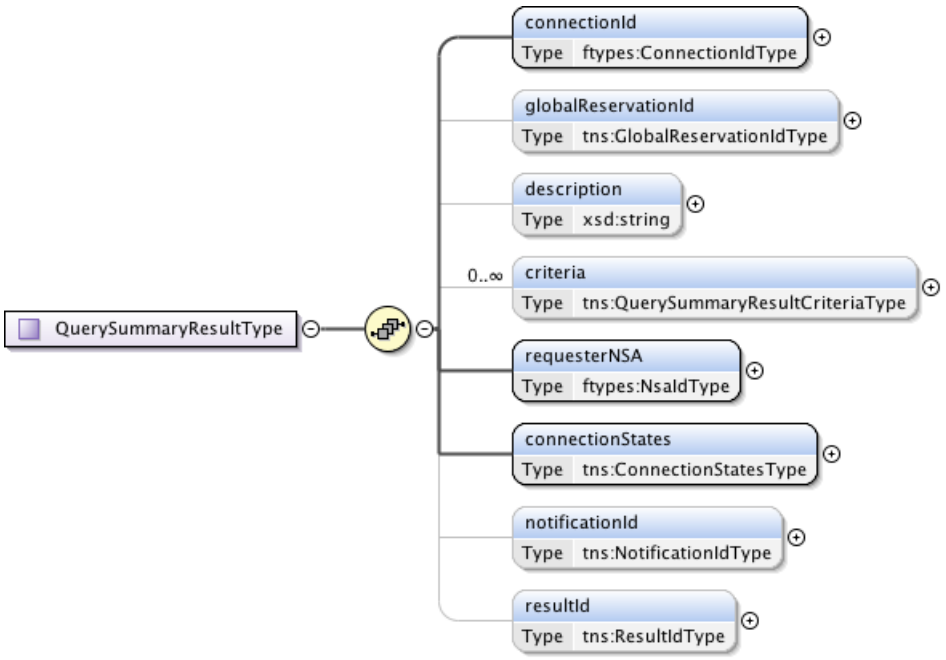

Figure 115 – *QuerySummaryResultType*.

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	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*.
resultId	O	If present will hold the result identifier of the most recent confirmed, failed, or error result against this reservation. The resultId can be used in the queryResult operation to retrieve the associated operation results.

Table 92 *QuerySummaryResultType* message parameters

8.5.1.28 *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.

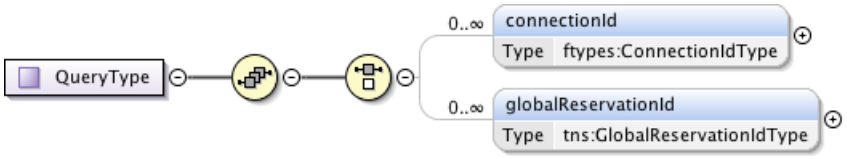

Figure 116 – *QueryType*.

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 93 *QueryType* message parameters

8.5.1.29 *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.

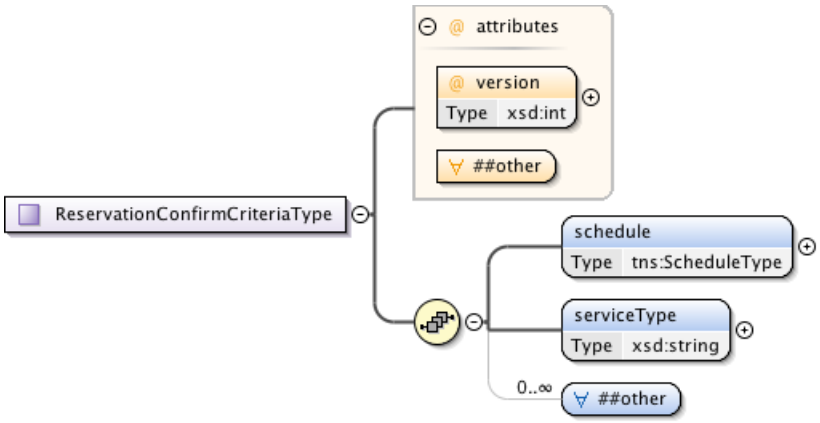

Figure 117 – *ReservationConfirmCriteriaType*.

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.
serviceType	M	The specific service type of this reservation. This service type string maps into the list of supported service definitions defined by the network providers. In turn, the service type specifies the specific service elements carried in an instance of this type (through the ANY definition) that are associated with the requested service.
any ##other	O	Provides a flexible mechanism allowing additional elements to be provided such as the service-specific attributes specified by *serviceType*. Additional 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 94 *ReservationConfirmCriteriaType* message parameters

8.5.1.30 *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.

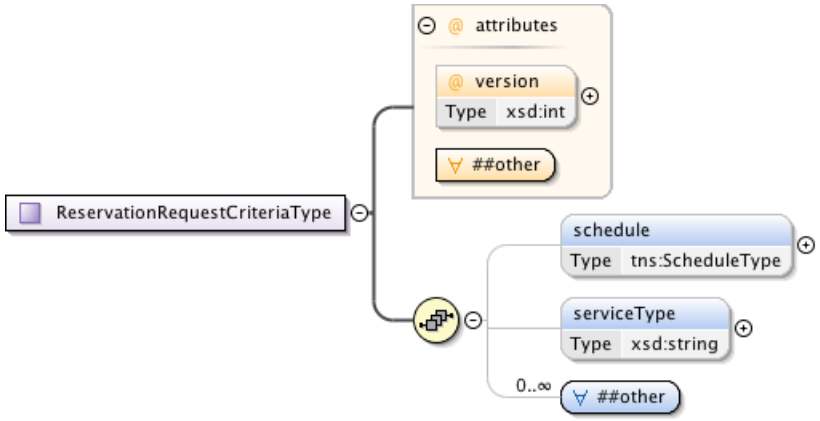

Figure 118 – *ReservationRequestCriteriaType*.

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	O	Time parameters specifying the life of the service. If not present then the service to start immediately and run for an infinite time.
serviceType	O	Specific service type being requested in the reservation. This service type string maps into the list of supported service definitions defined by the network providers, and in turn, to the specific service elements carried in this element (through the ANY definition) required to specify the requested service. The service type is mandatory in the original reserve request, and optional in a reserve issued to modify an existing reservation.
any ##other	O	Provides a flexible mechanism allowing additional elements to be provided such as the service-specific attributes specified by *serviceType*. Additional 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 95 *ReservationRequestCriteriaType* message parameters

8.5.1.31 *ReserveConfirmedType*

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

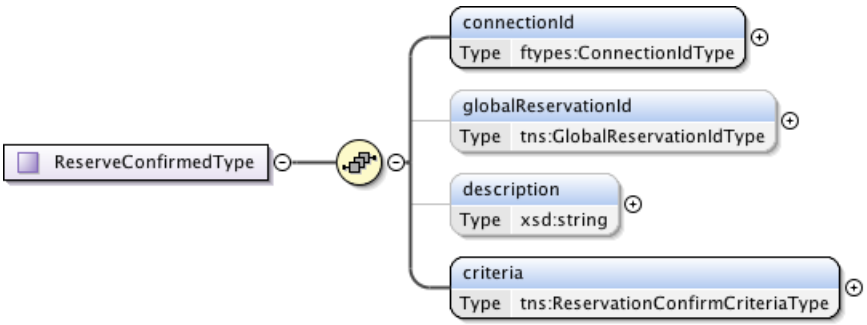

Figure 119 – *ReserveConfirmedType*.

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	M	Versioned reservation criteria information.

Table 96 *ReserveConfirmedType* message parameters

8.5.1.32 *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.

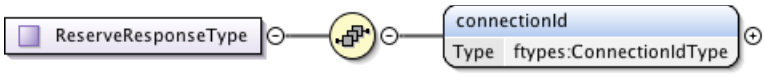

Figure 120 – *ReserveResponseType*.

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 97 *ReserveResponseType* message parameters

8.5.1.33 *ReserveTimeoutRequestType*

This is the type definition for the reserve timeout notification message. This is an autonomous message issued from a PA to an 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.

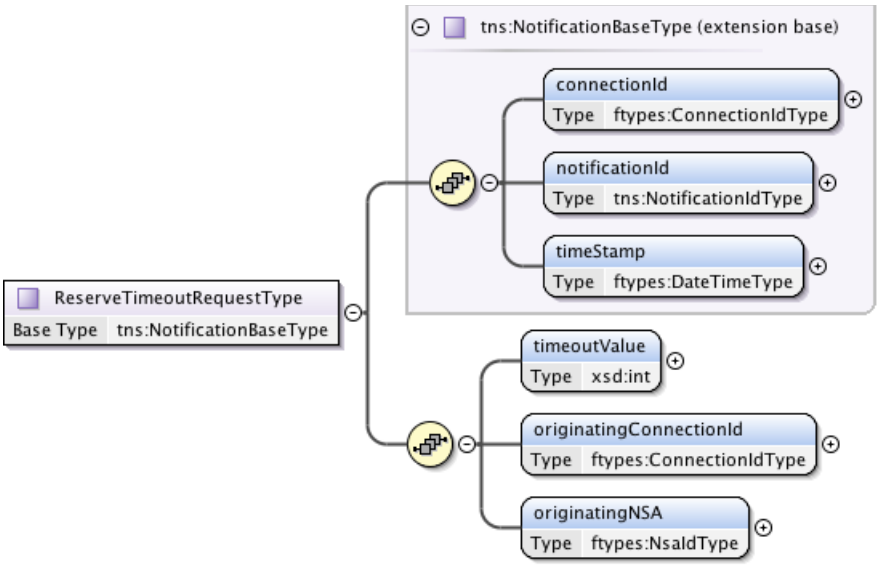

Figure 121 – *ReserveTimeoutRequestType*.

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 98 *ReserveTimeoutRequestType* message parameters

8.5.1.34 *ReserveType*

This is the type definition that models the reserve message that allows an 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.

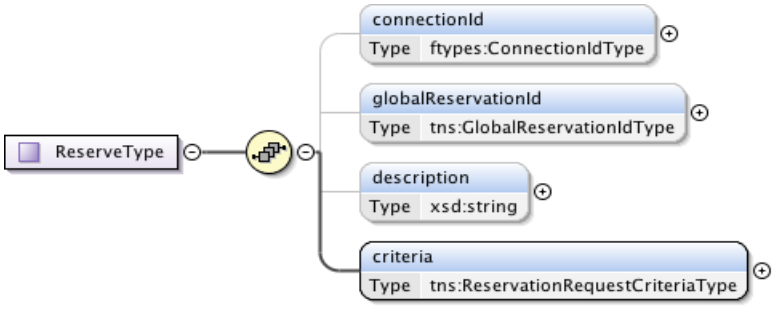

Figure 122 – *ReserveType*.

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 99 *ReserveType* message parameters

8.5.1.35 *ScheduleType*

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

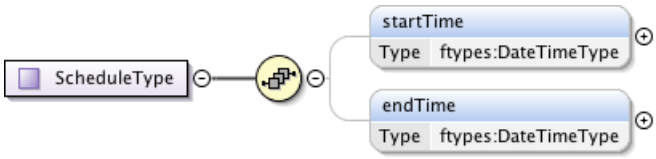

Figure 123 – *ScheduleType*.

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 100 *ScheduleType* message parameters

8.5.2 Simple Types

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

8.5.2.1 *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.

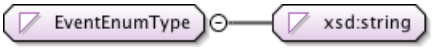

Figure 124 – *EventEnumType*.

8.5.2.2 *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 125 – *GlobalReservationIdType*.

8.5.2.3 *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.
- ***PassedEndTime*** - The reservation has exceeded scheduled end time.
- ***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 126 – *LifecycleStateEnumType*.

8.5.2.4 *NotificationIdType*

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

Figure 127 – *NotificationIdType*.

8.5.2.5 *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 for a reservation is created in the Released state when the first reserve request is received, however, a provision request cannot be processed until the first version of the reservation has been successful committed. If a provision request is received before the first version of a reservation has been created, then it must be rejected with an error.

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 128 – *ProvisionStateEnumType*.

8.5.2.6 *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 129 – *ReservationStateEnumType*.

8.5.2.7 *ResultIdType*

A specific type for a r*esultId* that is an identifier unique in the context of a *connectionId*.

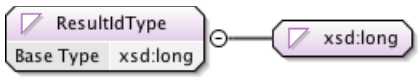

Figure 130 – *ResultIdType*.

9. Security

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

9.1 Transport Layer Security

TLS is used to ensure secure communication between NSAs. TLS also supports 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 and authentication via the X.509 certificates, and protects against replay attacks. Authorization is done at the NSAs application level. TLS version 1.0 MUST be supported. NSAs MAY use SSLv3 and TLS versions higher than 1.0 where possible.

9.2 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.

10. 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

11. Glossary

Activate	When provisioning of a Connection has been completed the Connection is considered to be Active. A dataPlaneStateChange notification is sent to the RA with “active” set to “true” 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 an 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 sent 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 an NSI message-based platform capable of supporting a suite of Network Services such as the Connection Service and the Topology Service.
NSI Message	An NSI Message is a structured unit of data sent between an RA and a PA.
NSI Topology	The NSI Topology defines a standard ontology and a schema to describe network resources that are managed to create the NSI service. The NSI Topology as used by the NSI CS (and in future other NSI services) is described in: GWD-R-P: Network Service Interface Topology Representation [3].

ero	An Explicit Routing Object (ero) is a parameter in a Connection request. It is an ordered list of STP constraints to be used by the inter-Network pathfinder.
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/Provider Agent (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 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 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.

12. 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.

13. 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.

14. Full Copyright Notice

15. 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 4.3.

	>rsv.rq	>rsvabort.rq	>rsvcommit.rq	<rsv.cf	<rsv.fl	<rsvabort.cf	<rsvcommit.cf	<rsvcommit.fl	(ReserveTimeout)
Reserve Start	***Reserve Checking***	Reserve Start	Reserve Start	Reserve Start	Reserve Start	Reserve Start	Reserve Start	Reserve Start	Reserve Start
	>rsv.rq	<rsvabort.na	<rsvcommit.na						
Reserve Checking	Reserve Checking	Reserve Checking	Reserve Checking	***Reserve Held***	***Reserve Failed***	Reserve Checking	Reserve Checking	Reserve Checking	Reserve Checking
	<rsv.na	<rsvabort.na	<rsvcommit.na	***<rsv.cf***	***<rsv.fl***				
Reserve Held	Reserve Held	***Reserve Aborting***	***Reserve Commiting***	Reserve Held	Reserve Held	Reserve Held	Reserve Held	Reserve Held	***Reserve Timeout***
	<rsv.na	***>rsvabort.rq***	***>rsvcommit.rq***						***<rsvtimeout.nt***
Reserve Committing	Reserve Committing	Reserve Committing	Reserve Committing	Reserve Committing	Reserve Committing	Reserve Committing	***Reserve Start***	***Reserve Start***	Reserve Committing
	<rsv.na	<rsvabort.na	<rsvcommit.na				***<rsvcommit.cf***	***<rsvcommit.fl***	
Reserve Failed	Reserve Failed	***Reserve Aborting***	Reserve Failed	Reserve Failed	Reserve Failed	Reserve Failed	Reserve Failed	Reserve Failed	Reserve Failed
	<rsv.na	***>rsvabort.rq***	<rsvcommit.na						
Reserve Aborting	Reserve Aborting	Reserve Aborting	Reserve Aborting	Reserve Aborting	Reserve Aborting	***Reserve Start***	Reserve Aborting	Reserve Aborting	Reserve Aborting
	<rsv.na	<rsvabort.na	<rsvcommit.na			***<rsvabort.cf***			
Reserve Timeout	Reserve Timeout	***Reserve Aborting***	***Reserve Start***	Reserve Timeout	Reserve Timeout	Reserve Timeout	Reserve Timeout	Reserve Timeout	Reserve Timeout
	<rsv.na	***>rsvabort.rq***	***<rsvcommit.fl***						

	Input message/Event	Illeagal request. Reply with "not applicable".
Current State	Next State	Non expected input. Should be an error
	Output message	

Table 101 RSM transition table

	>prov.rq	>rel.rq	<prov.cf	<rel.cf
Released	***Provisioning***	Released	Released	Released
	>prov.rq	<rel.na		
Provisioning	Provisioning	Provisioning	***Provisioned***	Provisioning
	<prov.na	<rel.na	***<prov.cf***	
Provisioned	Provisioned	***Releasing***	Provisioned	Provisioned
	<rsv.na	***>rel.rq***		
Releasing	Releasing	Releasing	Releasing	***Released***
	<prov.na	<rel.na		***<rel.cf***

	Input message	Illeagal request. Reply with "not applicable".
Current State	Next State	Non expected input. Should be an error
	Output message	

Table 102 PSM transition table

Lifecycle State Machine (LSM)

	>term.rq	<term.cf	<forcedEnd
Created	***Terminating***	Created	***Failed***
	>term.rq		***<forcedEnd***
Failed	Failed	***Terminated***	***Failed***
	<term.na	***<term.cf***	***<forcedEnd***
Terminating	Terminating	***Terminated***	***Terminating***
	<term.na	***<term.cf***	***<forcedEnd***
Terminated	Terminated	Terminated	***Terminated***
	<term.na		***<forcedEnd***

	Input message		Illeagal request. Reply with "not applicable".
Current State	Next State		Non expected input. Should be an error
	Output message		

Table 103 LSM transition table

A service exception MUST be immediately returned when an invalid message is detected. For example:

- In the case where a .na message is specified in these tables Service exception 201 ‘invalid message’ is returned.
- Undefined *connectionId* in the request will return a service exception 203.

An error response message is sent when the incoming message is valid but there are processing issues that need to be notified (e.g. a problem has been encountered during provisioning).

16. Appendix B: Error Messages and Best Practices

16.1 Error Messages

The following set of error codes SHOULD be used. Any of these service exceptions can be sent in either the SOAP fault reply to the original request, a failed reply message, or an error reply message.

errorId	***errorDescription***	***Text***	***variables***
00100	PAYLOAD_ERROR		
00101	MISSING_PARAMETER	Invalid or missing parameter	Include the parameter name that is missing.
00102	UNSUPPORTED_PARAMETER	A provided request parameter that MUST be processed contains an unsupported value.	Include the parameter name that is unsupported.
00103	NOT_IMPLEMENTED		Include the capability that is not implemented.
00104	VERSION_NOT_SUPPORTED	The service version requested in NSI header is not supported.	Return type *protocolVersion* and value the version requested.
00200	CONNECTION_ERROR		
00201	INVALID_TRANSITION	Connection state machine is in invalid state for received message.	Include the current state of the state machine.
00202	CONNECTION_EXISTS	Schedule already exists for *connectionId*.	
00203	CONNECTION_NONEXISTENT	Schedule does not exist for *connectionId*.	
00204	CONNECTION_GONE		

00205	CONNECTION_CREATE_ERROR	Failed to create connection (payload was ok, something went wrong)	
00300	SECURITY_ERROR		
00301	AUTHENTICATION_FAILURE		
00302	UNAUTHORIZED		
00400	TOPOLOGY_ERROR		
00403	NO_PATH_FOUND	Path computation failed to resolve route for reservation.	
00500	INTERNAL_ERROR	An internal error has caused a message processing failure.	
00501	INTERNAL_NRM_ERROR	An internal NRM error has caused a message processing failure.	Include information describing the specific NRM error.
00600	RESOURCE_UNAVAILABLE		
00700	SERVICE_ERROR	Reserved for service specific errors as defined by *serviceType* and the corresponding service definition.	

Table 104 Error messages

16.2 NTP servers

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

16.3 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 (AG) 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 2 minutes for reservations to a provider-only NSA, and longer for hierarchical requests to aggregator NSAs depending on the number of levels of recursion.

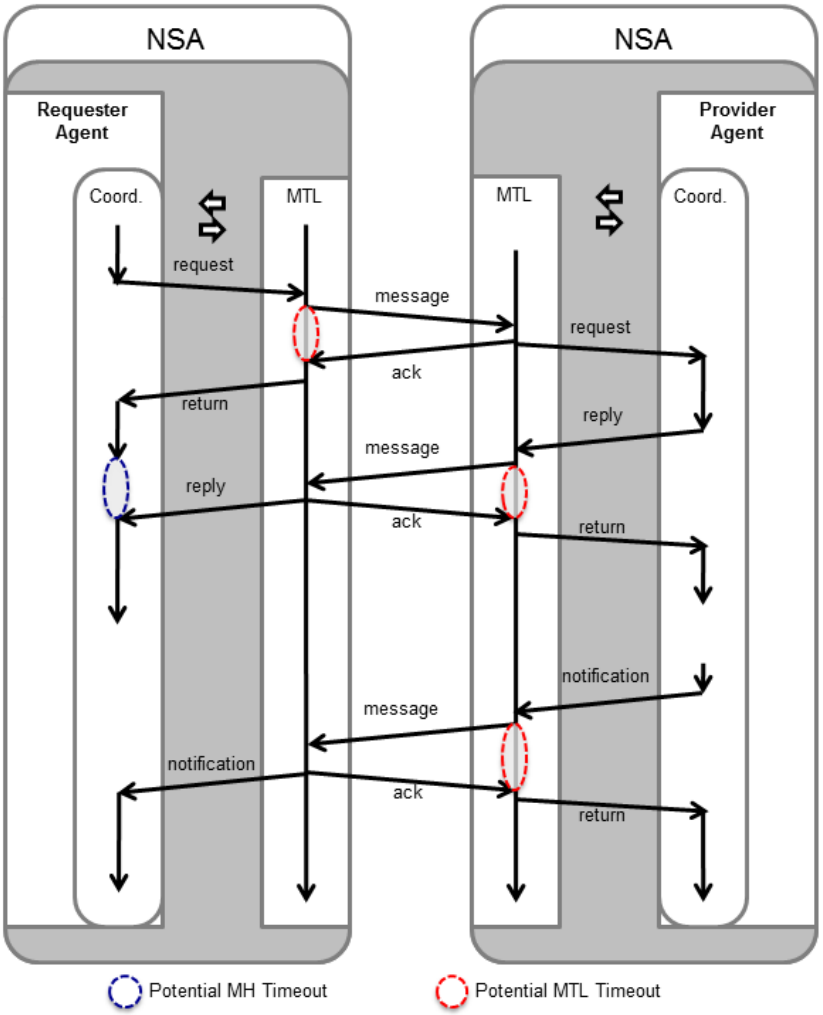

Figure 131: Potential MH/MTL timeout sequences

17. Appendix C: 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 132 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.

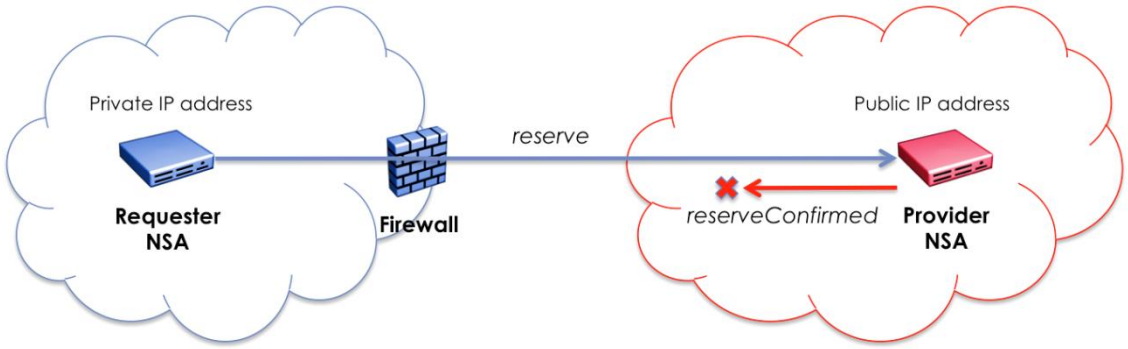

Figure 132 – RA behind a firewall with private IP address.

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 133** 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.

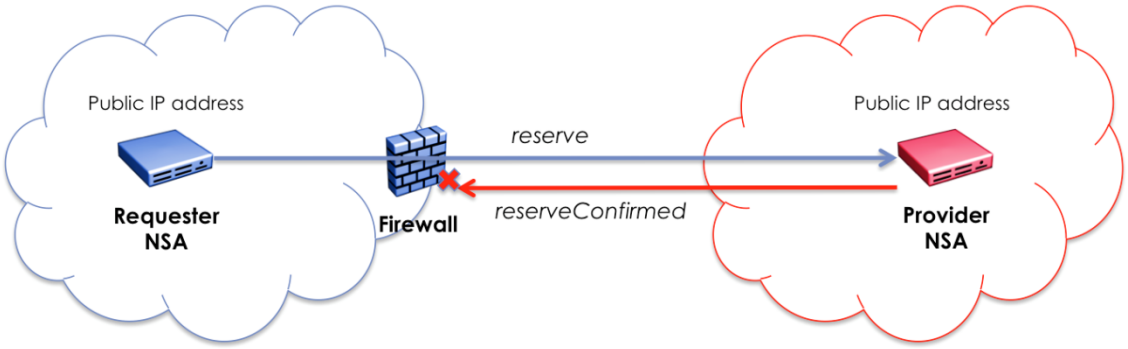

Figure 133 – RA behind a firewall with public IP address.

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

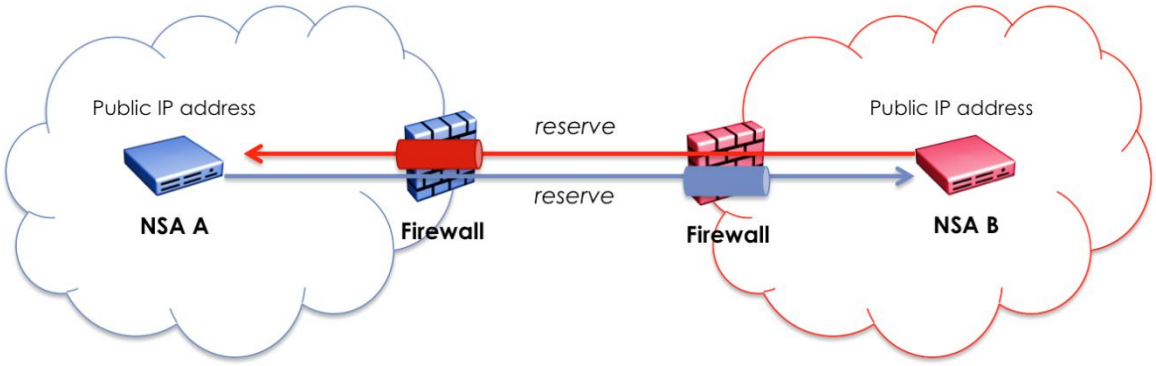

Figure 134 – Peer NSA behind a firewall with public IP addresses.

A slightly more complicated NSA deployment occurs when one or both of the peer NSAs 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 that an 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.

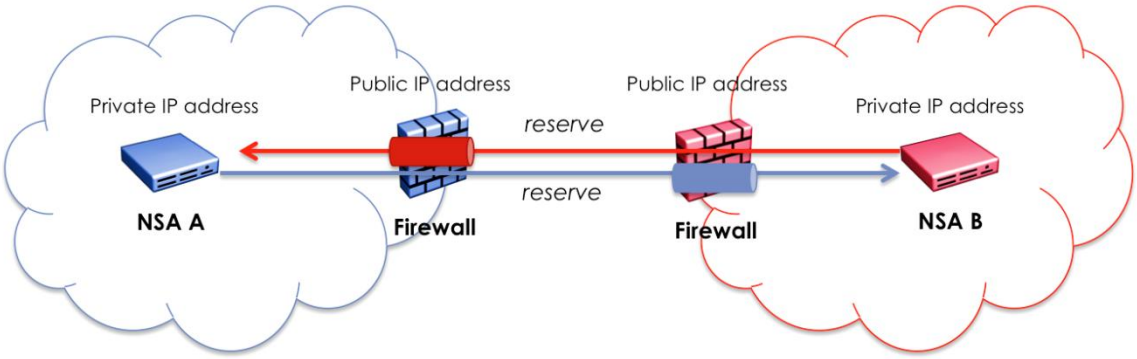

Figure 135 – Peer NSA behind a firewall with private IP addresses.

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

In NSI CS a simple set of synchronous operations have been added to allow an RA isolated behind a firewall to interact with a PA supporting a publically accessible interface. These synchronous operations will block until the confirmed, failed, or error message is available and return it in the results of the synchronous request (where the current asynchronous operations return an ACK).

The existing *reserve*, *reserveCommit*, *reserveAbort*, *provision*, *release*, and *terminate* operation sets have been modified to accept requests without a *replyTo* parameter within the NSI CS header.

These operations will behave normally, except a confirmed/failed response will not be sent back to the RA. This behavior is triggered by the lack of a *replyTo* parameter. It is the responsibility of an RA to determine the result of these operations through changes to state machines associated with the reservation via the firewall safe *querySummarySync* operation. Results of a previously issued operation can be determined by polling state machines associated with the reservation.

NS CS version 2.0 also introduced additional modeling of event notifications and operation results against reservations to help support a synchronous polling RA. A notification identifier and result identifier has been added to the reservation query information to indicate a notification/result has been received against that reservation. Without the ability to receive asynchronous notifications/results, these synchronous polling RA can use the new firewall safe *queryNotificationSync* operation to retrieve a list of notifications against the reservation, or the *queryResultSync* operation to retrieve a list of operation results against the reservation.

To summarize, with the optional *replyTo* parameter, the introduction of notification modeling within a reservation, and the firewall safe *querySummarySync* , *queryNotificationSync, and queryResultSync* operations, it is possible to build a fully functional firewall-safe RA.

18. Appendix D: 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 a uPA, these are described separately.

18.1 Aggregator NSA

18.1.1 Processing of NSI Requests

The following outlines the messages received by the AG's Coordinator from external NSAs (e.g. parent or child NSAs), and the corresponding interactions between the Coordinator and various internal state machine functions.

```
NSI_rsv.rq(Conn_ID, Corr_ID, Ver)  /* from parent NSA */
    if (new Conn_ID) then
    {
        create state machine RSM(Conn_ID)  /* initial state = ReserveStart */
        create state machine LSM(Conn_ID)  /* initial state = Created */
        create state machine PSM(Conn_ID)  /* initial state = Released */
        do pathfinding -> create entry for all children in
          connection_segment_list(Conn_ID, Child_NSA)
    }
    send rsv.rq(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  /* 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)
    }
```

18.1.2 Requests from State Machines

The following outlines the messages received by the AG's Coordinator from internal state machine functions, and the corresponding actions and messages to external NSAs (e.g. parent or child NSAs).

```
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
```

18.2 Ultimate PA

18.2.1 Processing of NSI Requests

The following outlines the messages received by the uPA's Coordinator from external NSAs (e.g. parent NSAs), and the corresponding interactions between the Coordinator and various internal state machine functions.

```
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 */
```

18.2.2 Requests from State Machines

The following outlines the messages received by the uPA's Coordinator from internal state machine functions, and the corresponding actions and messages to external NSAs (e.g. parent NSAs).

```
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
```

19. Appendix E: Service-Specific Schema

One of the primary objectives of NSI CS is to remove the dependencies of data plane service specification from the core NSI CS protocol (this is new in NSI CS v2.0 compared to earlier versions of NSI). This decoupling allows the existing NSI CS protocol to remain stable while permitting changes to the services offered by a network provider without impacting the existing protocol. This section documents the decoupled Point-to-Point Service Schema.

19.1 Restructuring *criteria* element

In NSI CS 2.0 the Point-to-Point service-specific *capacity*, *path*, and *serviceAttributes* elements are removed from the *criteria* element, used for example in the reserve message elements. These Point-to-Point service-specific elements are repackaged into a separate service-specific schema definition, which is allocated a dedicated namespace for use when referencing the contained elements. The *criteria* element was extended to include an *ANY* child element allowing generic inclusion of external service schemas. In addition to the service specification decoupling, the CS uses an element called *serviceType*, which is described in the next section. These criteria are shown in Section 8.5.1.30.

19.2 The *serviceType* element

The *serviceType* element names the specific service type requested in the reservation. This service type string maps to a specific Service Definition template defined by the network providers describing the type of service offered, parameters supported in a reservation request (mandatory and optional), defaults for parameters if not specified (as well as maximums and minimums), and other attributes relating to the service offering. The NSA in turn uses this information to determine the specific service parameters carried in the *criteria* element as part of the reservation request.

The Service Definition template is an important component in the solution, linking the opaque information carried in the NSI CS protocol to the concrete parameters needed to satisfy a specific service request.

19.3 Service-specific errors

The NSI CS protocol commonly utilizes the *ServiceExceptionType* structure to convey error information associated with SOAP faults, failed messages, and error messages. The structure is extremely flexible and able to handle both simple high-level error information, as well as detailed errors down to the individual attribute value causing a problem. The current *ServiceExceptionType* is defined in Section 8.3.1.

The NSI CS protocol uses a hierarchal error code structure to group related error codes together under a common parent error code value. A service-specific parent error code *SERVICE_ERROR*(00700) has been defined for use by individual service specification. As new services are offered, and existing ones modified, these service-specific errors can be modified as needed with no impact on the core NSI CS protocol.

Context for these service-specific errors is provided by the *serviceType* element included in the *ServiceExceptionType* structure returned when an NSA generates a service-specific error. This *serviceType* element maps into the service definition used for the service request[1] on this failed segment and, in turn, to a detailed description of the service-specific error. Table 105 shows the service-specific errors defined for the basic point-to-point service.

text	errorId	Description
SERVICE_ERROR	00700	Parent error classification for a service-specific error.
UNKNOWN_STP	00701	Could not find STP in topology database.
STP_RESOLUTION_ERROR	00702	Could not resolve STP to a managing NSA.
VLANID_INTERCANGE_NOT_SUPPORTED	00703	VLAN interchange not supported for requested path.
STP_UNAVALABLE	00704	Specified STP already in use.
CAPACITY_UNAVAILABLE	00705	Insufficient capacity available for reservation.

Table 105 – NSI-CS point-to-point service-specific errors.

19.4 Point-to-point service-specific schema

All service capabilities of earlier versions of the NSI CS have been captured in the service-specific schema for NSI CS. Service parameters must be encapsulated in an XML element for inclusion in the *criteria* element of a reservation request. In addition, any modifiable parameters of the reservation must also be defined as XML elements for inclusion in the *criteria* element of a modification request.

Namespace definition: http://schemas.ogf.org/nsi/2013/12/services/point2point

19.4.1 Service Elements

19.4.1.1 *p2ps*

This point-to-point service element is used within the *criteria* element to specify a generic point-to-point service request in the NSI CS protocol. It provides functional equivalent to the point-to-point service integrated in earlier versions of NS CS, and can be used for point-to-point Ethernet service offerings.

[1] The *serviceType* is included since the original *serviceType* specified in the *reserve* request may have been re-mapped into a different *serviceType* when sent to a child NSA.

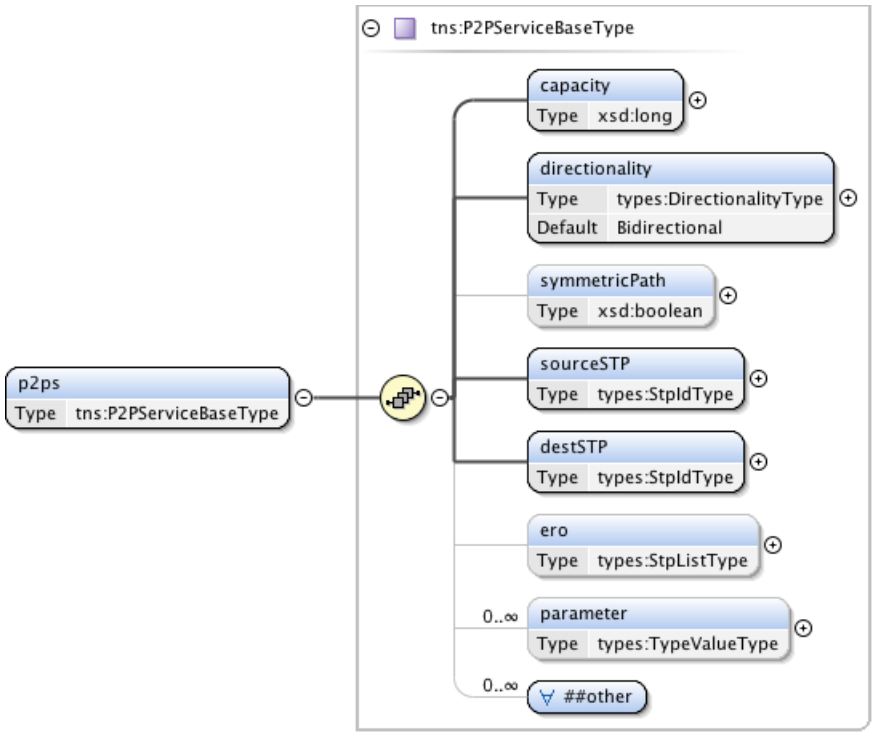

Figure 136 – *p2ps* service element.

Parameters

The *p2ps service element* has the following parameters:

Parameter	Description
capacity	Capacity of the service. Units for the capacity parameter are defined in the associated service definition.
directionality	The (uni or bi) directionality of the service.
symmetricPath	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	Source STP identifier of the service.
destSTP	Destination STP identifier of the service.
ero	A hop-by-hop ordered list of STPs from sourceSTP to destSTP representing a path that the connection must follow. This list does not include sourceSTP or destSTP.
parameter	A flexible non-specific parameters definition allowing for specification of parameters in the Service Definition that are not defined directly in the service-specific schema.
##other	For future expansion and extensibility.

Table 106 *p2ps service element* parameters

19.4.1.2 *capacity*

The *capacity* element is defined for a modification of the capacity of an existing service. The unit of capacity is specified in the Service Definition associated with the requested service.

Figure 137 – *capacity* service element.

19.4.1.3 *parameter*

The *parameter* element, as a member of the *p2ps* service element, is used to add additional service parameters not explicitly defined in the schema, but specified in the Service Definition.
The *parameter* element is specified individually within the *criteria* element when a modification to one of these Service Definition defined parameters is required in an existing reservation.

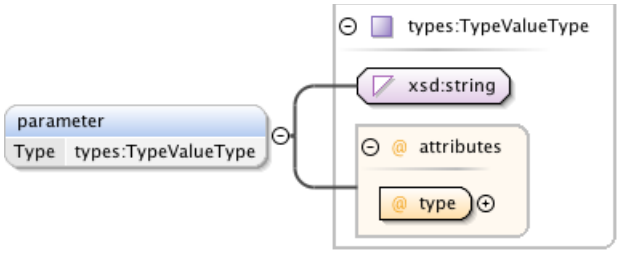

Figure 138 – *parameter* service element.

19.4.2 Complex Types

These complex type definitions are utilized by the service-specific schema element definitions.

19.4.2.1 *P2PServiceBaseType*

The *P2PServiceBaseType* is a structure for a generic point-to-point service specification. At the moment this type supports a unidirectional or bidirectional service.

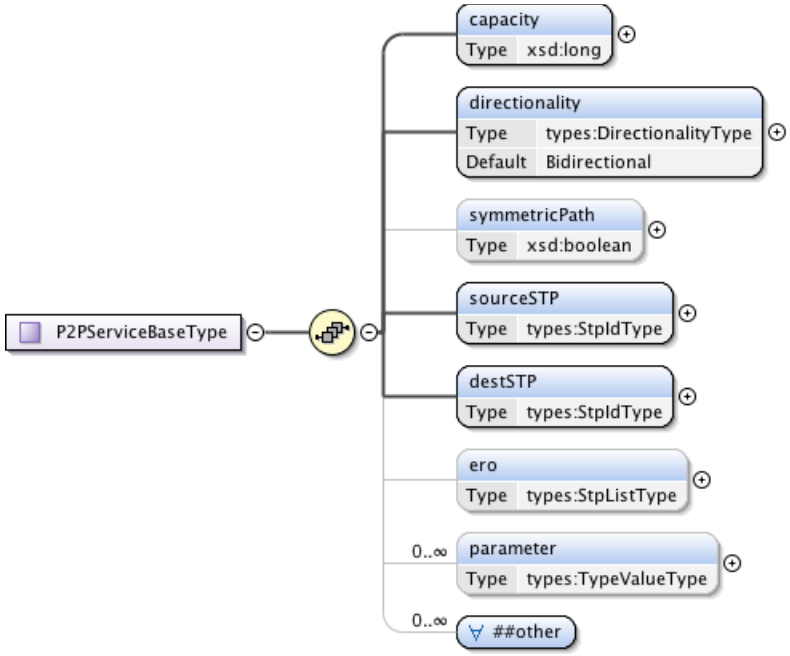

Figure 139 – *P2PServiceBaseType*.

Parameters

The *P2PServiceBaseType* has the following parameters (M = Mandatory, O = Optional):

Parameter	M/O	Description
capacity	M	Capacity of the service. Units for the capacity parameter are defined in the associated service definition.
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 identifier of the service.
destSTP	M	Destination STP identifier of the service.
ero	O	A hop-by-hop ordered list of STP from sourceSTP to destSTP representing a path that the connection must follow. This list does not include sourceSTP or destSTP.
parameter	O	A flexible non-specific parameters definition allowing for specification of parameters in the Service Definition that are not defined directly in the service specific schema.
##other	O	For future expansion and extensibility.

Table 107 *P2PServiceBaseType* parameters.

19.5 Generic Service Types

These are generic service type definitions that can be used to build service-specific schema. These definitions are currently used by the point-to-point service definitions.

Namespace definition: http://schemas.ogf.org/nsi/2013/12/services/types

19.5.1 Complex Types

These complex type definitions are utilized by the service-specific schema complex type definitions.

19.5.1.1 *OrderedStpType*

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

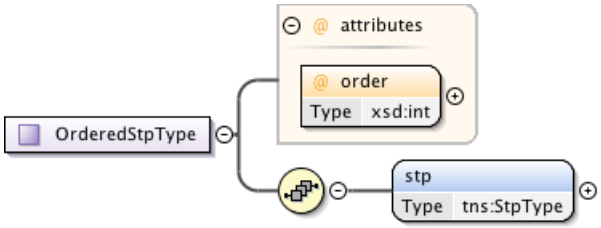

Figure 140 – *OrderedStpType*.

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 108 *OrderedStpType* parameters.

19.5.1.2 *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.

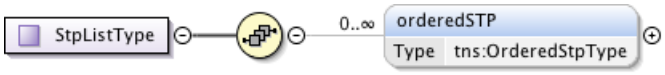

Figure 141 – *StpListType*.

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 109 *StpListType* message parameters

19.5.2 Simple Types

These simple type definitions are utilized by the service-specific schema complex type definitions.

19.5.2.1 *StpIdType*

This is the Service Termination Point (STP) identifier type used in a service request for identifying endpoints in path selection.

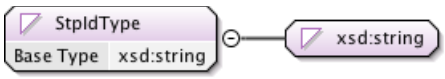

Figure 142 – *StpIdType*.

19.5.2.2 *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.

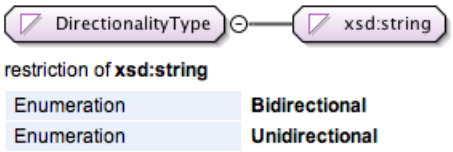

Figure 143 – *DirectionalityType*.

19.6 Reservation request

Here is an example *reserve* request XML message for a bidirectional service as defined in NSI CS version 2.0. There are a few things to note:

- The *serviceType* element is added to identify the desired service requested and will identify the specific service elements carried in *criteria*. (in this case the *p2ps* element).
- The ***p2p*** namespace is defined in the *reserve* element using a unique URL defining the service XSD document. All types needed for this point-to-point service in that XSD document.
- The *p2ps* element is included in the *criteria* element and includes all service-specific parameters.

```
<nsi:reserve xmlns:nsi="http://schemas.ogf.org/nsi/2013/12/connection/types"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p2p="http://schemas.ogf.org/nsi/2013/12/services/point2point">

  <connectionId>urn:uuid:4b4a71d0-3c71-47cf-a646-beacb14a4c72</connectionId>
  <globalReservationId>urn:uuid:83fe4f36-5b38-41b6-bc46-a362a06a54ee</globalReservationId>
  <description>My example reservation using NSI CS 2.0.</description>
  <criteria version="1">
    <schedule>
      <startTime>2013-12-30T09:30:10Z</startTime>
      <endTime>2013-12-30T10:30:10Z</endTime>
    </schedule>
    <serviceType>http://services.ogf.org/nsi/2013/12/descriptions/EVTS.A-GOLE</serviceType>
    <p2p:p2ps>
      <capacity>1000</capacity>
      <directionality>Bidirectional</directionality>
```

```
        <symmetricPath>true</symmetricPath>
        <sourceSTP>urn:ogf:network:netherlight.net:2012:uvalight-netherlight</sourceSTP>
        <destSTP>urn:ogf:network:netherlight.net:2012:netherlight-czechlight</destSTP>
        <parameter type="mtu">9500</parameter>
      </p2p:p2ps>
    </criteria>
</nsi:reserve>
```

19.7 Reservation modification

For a base point-to-point service specification we support the modification of *schedule* (start or end time), as well as the *capacity* of the service. The *schedule* element is within the core *criteria* element, and remains as is, specifying a change in the combination of *startTime* and *endTime* as desired. For the external service schema, only the elements to be modified are included in the request. These will be defined as separate elements within their schema definition for inclusion as modifiable items.

Below is an example *reserve* modification request XML message where we are requesting a modification to the *capacity* parameter of the reservation. Notice the *serviceType* element is not required since the reservation is already bound by the original *serviceType* specified in the *reserve* request.

```
<nsi:reserve xmlns:nsi="http://schemas.ogf.org/nsi/2013/12/connection/types"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p2p="http://schemas.ogf.org/nsi/2013/12/services/point2point">

    <connectionId>urn:uuid:4b4a71d0-3c71-47cf-a646-beacb14a4c72</connectionId>
    <criteria version="2">
      <p2p:capacity>500</p2p:capacity>
    </criteria>
</nsi:reserve>
```

20. Appendix F: Tree and Chain Connection Examples

20.1 Connection managed by an NSA chain

Figure 144 shows an example of a Connection managed by an NSA chain. Each NSA is associated with one Network as an NSA/Network pairing. In this case the Connection request is forwarded between NSAs in the same sequence as the Connection transits the Networks.

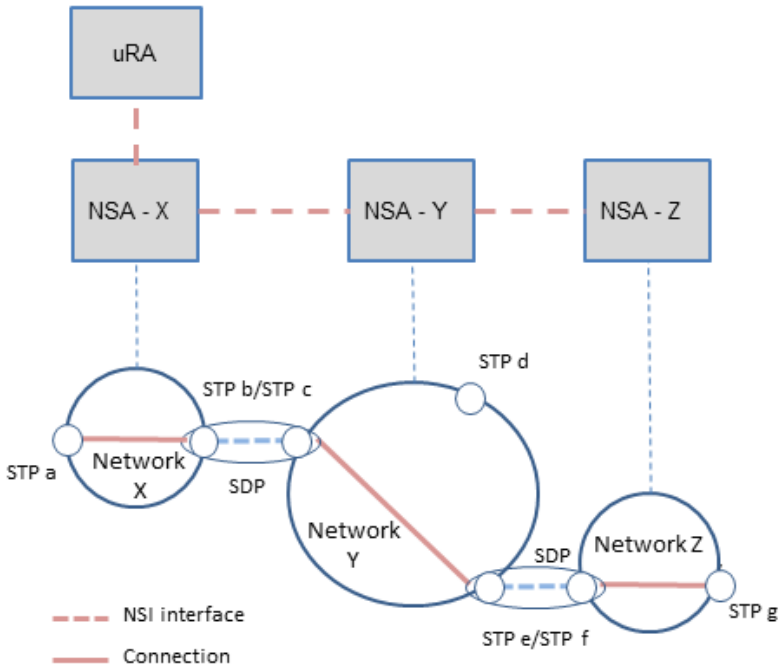

Figure 144: Example of Connection managed by an NSA chain

This example shows an NSI 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), and Network Z (STP f, STP g)

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

Assuming a Connection request comes from the uRA to NSA-X to reserve a Connection STP a to STP g, then NSA-X will perform pathfinding on the topology and determine that, to make this Connection, NSA-X needs to 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.

20.2 Connection managed by an NSA tree

Figure 145 shows an example of a Connection managed by an NSA tree. In this case the NSI message is forwarded between NSAs in a different sequence compared to the sequence in which the Connection transits the Networks.

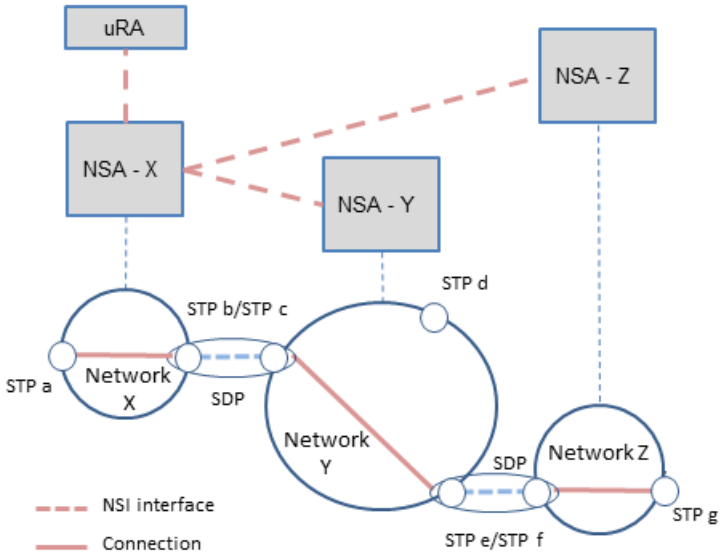

Figure 145: Example of a Connection managed by a NSA tree

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

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

Assuming a Connection request comes from the uRA to NSA-X to reserve a Connection from STP a to STP g, then NSA-X will perform pathfinding on the topology and determine that, to make this Connection, NSA-X needs to reserve a local Connection from STP a to STP b. Next NSA-X forwards a requestto NSA-Y to connect STP c to STP e, and to NSA-Z to connect STP f to STP g. . NSA-Y builds its local Connection STP c to STP e and NSA-Z builds its local Connection STP f to STP g. In this scenario, NSA-X is responsible for stitching Network Y and Network Z together at the SDP made up of STP e/STP f. This is because NSA-Y and NSA-Z will not communicate directly with one another.

21. References

1. OGF GWD-R-P “Network Service Framework v2.0”
2. OGF GWD-I Network Service Interface Topology Service Distribution Mechanisms https://redmine.ogf.org/dmsf_files/12980?download=
3. GWD-R-P Network Service Interface Topology Representation
4. OGF GFD.206: Network Markup Language Base Schema version 1 http://www.gridforum.org/documents/GFD.206.pdf
5. IETF RFC 5905, Network Time Protocol Version 4: Protocol and Algorithms Specification
6. IETF RFC 4122, A Universally Unique IDdentifier (UUID) URN Namespace
7. 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
8. 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

9. IETF RFC 4655, "A Path Computation Element (PCE)-Based Architecture", http://www.rfc-editor.org/rfc/rfc4655.txt
10. ISO 8601:2000 “Data elements and interchange formats — Information interchange — Representation of dates and times” or xsd dateTime
11. IETF RFC 5905, “Network Time Protocol Version 4: Protocol and Algorithms Specification”, http://tools.ietf.org/html/rfc5905
12. IETF RFC 6453, “A URN Namespace for the Open Grid Forum (OGF)”, http://tools.ietf.org/html/rfc6453
13. OGF GFD-CP.191 "Procedure for Registration of Subnamespace Identifiers in the URN:OGF Hierarchy”, http://www.ogf.org/gf/docs/
14. W3C XML “Schema Definition Language (XSD) 1.1 Part 2: Datatypes”, http://www.w3.org/TR/xmlschema11-2/#anyURI