034115

PHOSPHORUS

Lambda User Controlled Infrastructure for European Research

Integrated Project

Strategic objective:
Research Networking Testbeds

Deliverable reference number D1.4

Definition and development of the Network Service Plane and northbound interfaces development

Due date of deliverable: 2007-11-30
Actual submission date: 2007-11-30
Document code: Phosphorus-WP1-D1.4

Start date of project:
October 1, 2006

Duration:
30 Months

Organisation name of lead contractor for this deliverable:
Rheinische Friedrich-Wilhems-Universität Bonn

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract

This deliverable describes the Network Service Plane (NSP) and its interfaces, specifically its northbound interfaces towards a Meta-Scheduling System [MSS] or towards user client software. The NSP is responsible for coordinating the reservation of network resources that belong to different administrative domains which are managed by different Network Resource Provisioning Systems (NRPSs).

This deliverable specifies the reservation interface used for multidomain reservation management and the administrative topology interface used to manage the participating domains and the interdomain topology of the network. It describes the different system modules, and describes the functionality of the system as a whole by explaining the sequence of actions taken for different common workflows.

Project:	Phosphorus
Deliverable Number:	D1.4
Date of Issue:	30/11/07
EC Contract No.:	034115
Document Code:	Phosphorus-WP1-D1.4

Table of Contents

Project:	Phosphorus
Deliverable Number:	D1.4
Date of Issue:	30/11/07
EC Contract No.:	034115
Document Code:	Phosphorus-WP1-D1.4

Table of Figures

Table of Tables

0 Executive Summary

This deliverable defines the architecture and the functionality of the Network Service Plane (NSP) and its northbound interfaces towards the user or towards a Meta-Scheduling System (MSS). The main task of the NSP is to coordinate multiple network domains controlled by different administrative authorities such that this multidomain aspect is hidden towards the user.

1 Network Service Plane requirements

1.1 Task to be solved within the Network Service Plane

The Meta-Scheduling System (MSS) developed in WP3 is responsible for the co-allocation of Grid and network resources managed by different administrative entities. It is currently implemented in the UNICORE middleware, but it is designed as middleware independent as possible. The MSS does not distinguish between Grid and network resources. It treats the network in the same way as grid resource, whose availability can be queried and that can be reserved in the same way as processing power on a computing cluster. In a single domain environment, a Network Resource Provisioning System (NRPS) offers these services to the MSS. A NRPS is a system that has full knowledge about the underlying network's topology and the utilization of resources at different points in time.

The "Network Service Plane" (NSP) is a solution for a "multidomain environment" with several coexisting NRPSs, each of them controlling an own domain. The domains are assumed to be connected via interdomain links. In such an environment, routing decisions must also be made on an interdomain level, and several NRPSs must be queried to check resource availability and to make resource reservations. Therefore, an entity is required that is aware of network topology aspects and that offers an interface which hides the multidomain nature of the underlying network towards a MSS or towards a human user.

It is important to keep in mind that the NSP is not simply an NRPS at a higher level. The actual resource utilization management is done at the NRPS layer. Each NRPS represents an autonomous system that allows a limited view on the internal structure of the domain. The NSP can rather be seen as a subsystem of the MSS, since its main task is to coordinate the different NRPSs.

1.2 Required functionality

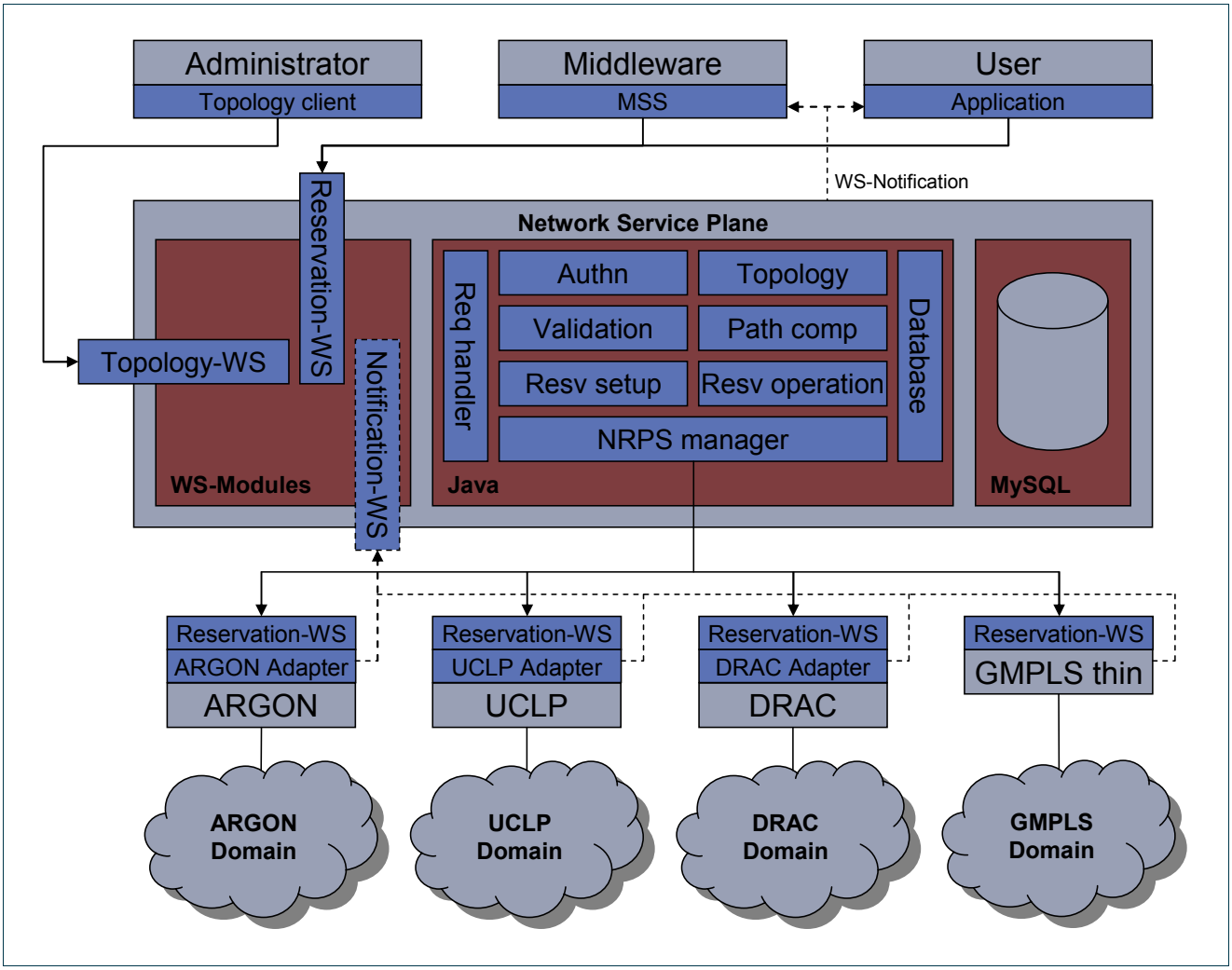

Figure 1.1: Overview over NSP interfaces and modules

The NSP and its role in the context of the Phosphorus project are outlined in Figure 1.1.

To achieve the task outlined above (Section 1.1), the NSP provides a user interface that offers services similar to a NRPS interface. This interface is called the “Reservation Webservice” or “Reservation-WS” (cf. Section 2.1). The Reservation-WS offers operations to query resource availability, to create and to cancel reservations, and to query a reservation’s status.

Essentially, multidomain requests received by the NSP have to be mapped to appropriate singledomain requests. However, a multidomain architecture introduces many degrees of freedom and therefore, a single multidomain request may also trigger a multitude of singledomain requests for each of the underlying domains. E.g., there can be many different interdomain paths that can be used to fulfill a network resource reservation request, and the NSP has to check the resource availability along these paths until a path with enough free resources is available to finally make a reservation (cf. Section 4.3).

Additionally, the NSP requires functionality to add, delete and edit domains and interdomain links. This functionality is accessed through a separate interface called the “Topology Webservice” or “Topology-WS” (cf. Sections 2.2, 4.1). Note that in general, the NSP has a rather limited view of the underlying domains, and the only topological data exchanged are the IDs of available endpoints located within the domains.

To efficiently implement fast responses to error situations, the NSP can also implement a “Notification Webservice” (“Notification-WS”). Adhering to the WS-notification standard [WSN], this allows a domain to inform the NSP e.g. that a reservation cannot be sustained because of a hardware failure. If this message is received from a transit domain, the NSP can try to establish alternative reservations; the user or the middleware process requesting the service would not need to be involved. In other cases, related reservations might be cancelled, and the user or middleware process could be informed through their Notification-WS interfaces.

2 Northbound interfaces

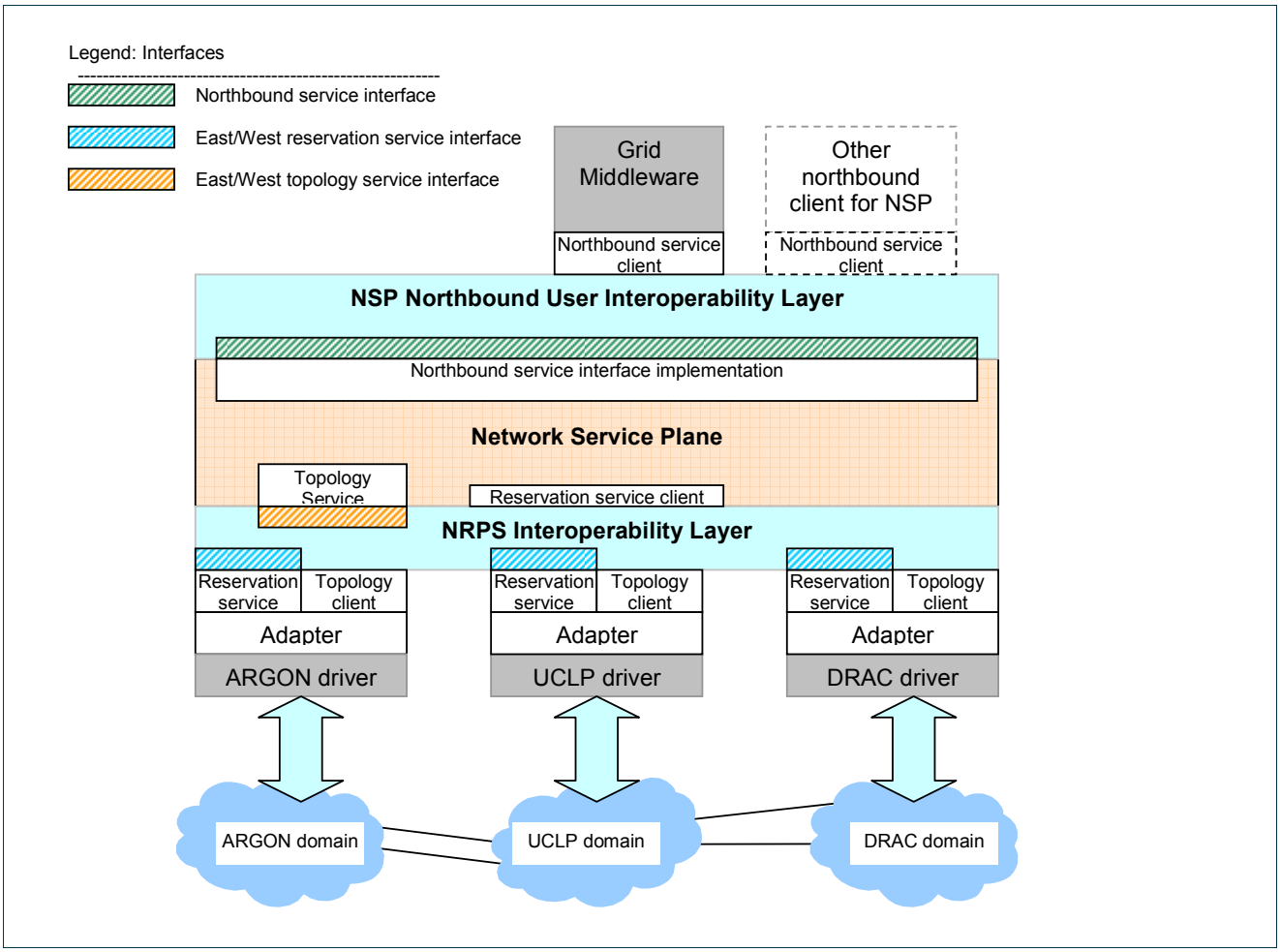

Figure 2.1: Overview over NSP interfaces and modules

The northbound interfaces disclose NSP functionality towards the higher level entities. The main northbound interface, the reservation interface, allows for requesting end-to-end resource provisioning that may span multiple domains. The reservation interface is described in Section 2.1.

Additionally to reservation management, the northbound interface allows for querying and modifying the domain topology. The topology interface is described in Section 2.2.

The data types used on the northbound interface are introduced in Section 2.3.

2.1 Reservation interface

The reservation interface is essentially the same as the interface that allows for NRPS interoperability, i.e. the interface that integrates an individual NRPS into the NSP. This is the case because both the NSP and the NRPSs allow resource provisioning, however, the functionality provided by the reservation interface of the NSP is of a higher level coordinative nature. Typically, within the reservation interface the individual domain resource reservations are handled and presented to the higher level entities as one set of reservations.

In the following subsections, data types are defined as **data type name : (composite) data type**. Underlined (composite) types are defined in Section 2.3. Not underlined data types are XML schema defined data types.

2.1.1 Reservation management

WSDL Operation Name	**getReservation**
Description	Retrieves the input by which a reservation request was made
Input parameters (XML type)	ReservationIdentifierType : long ServiceIdentifierType : integer
Output parameters (XML type)	GetReservationResponse : GetReservationResponseType
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	**getReservations**
Description	Retrieves all existing reservations for the specified period of time
Input parameters (XML type)	PeriodStartTime : dateTime PeriodEndTime : dateTime
Output parameters (XML type)	*Sequence of* Reservation : GetReservationsComplexType

Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	**cancelReservation**
Description	Cancels a network resource reservation
Input parameters (XML type)	ReservationIdentifierType : long
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	**getStatus**
Description	Returns the status of a service
Input parameters (XML type)	Service : ServiceConstraintType JobIdentifierType : long
Output parameters (XML type)	ServiceStatus : ServiceStatusType
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

2.1.2 Reservation setup

WSDL Operation Name	**isAvailable**
Description	Checks whether the specified service is available
Input parameters (XML type)	Service: ServiceConstraintType JobIdentifierType : long
Output parameters (XML type)	DetailedResult : ConnectionAvailabilityType *optional* AlternativeStartTimeOffset : long
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

Fault (XML type)	EndpointNotFoundFault : EndpointNotFoundFault

WSDL Operation Name	createReservation
Description	Creates the reservation of a path between 2 endpoints considering the specified constraints
Input parameters (XML type)	Service : ServiceConstraintType JobIdentifierType : long NotificationConsumerURL : string
Output parameters (XML type)	JobIdentifierType : long ReservationIdentifierType : long Detailed result : ConnectionAvailabilityType
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault
Fault (XML type)	EndpointNotFoundFault : EndpointNotFoundFault

2.1.3 Connection management

WSDL Operation Name	activate
Description	Activates a service
Input parameters (XML type)	ReservationIdentifierType : long ServiceIdentifierType : integer
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	bind
Description	Create binding between NRPS endpoint and application endpoint
Input parameters (XML type)	ReservationIdentifierType : long ServiceIdentifierType : integer ConnectionIdentifierType : integer EndpointID : EndpointIdentifierType
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault

Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

2.1.4 Job management

WSDL Operation Name	completeJob
Description	Modifies all *pre*-reservations belonging to the job to permanent reservations
Input parameters (XML type)	JobIdentifierType : long
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	cancelJob
Description	Cancels all reservations in the job. Ie. all resources are freed.
Input parameters (XML type)	JobIdentifierType : long
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

2.1.5 Other

WSDL Operation Name	getFeatures
Description	Retrieves information about the supported features of the NSP / NRPS
Input parameters	None
Output parameters (XML type)	*sequence of* FeatureName : string
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

2.2 Topology interface

This section contains the formal description of the topology interface operations and the types used within its operations. The topology interface supports adding of domains, domain endpoints and links between domain endpoints to the NSP thus enabling the NSP to compile a multidomain spanning topology. Also the topology interface supports editing and removal of domains, endpoints and links once they are added and known within the NSP in order to keep the topology known to the NSP up to date. Finally the topology known to the NSP can be retrieved through the *getDomain*, *getEndpoints* and *getLinks* operations.

In the current NSP implementation, interdomain links are assumed to be static links between border endpoints located in different domains. Further, it is assumed that they cannot be further divided into subchannels, so an interdomain link is only available for a single connection at a certain point in time. I.e., an interdomain link can be thought of as a specific wavelength. This reflects the current nature of the testbed. It therefore is not necessary that the NSP is aware of technological details of the interdomain links. However, in the future, the information model will be extended to allow for interdomain links with subchannels and to take technological details into account.

Also, domains are currently not required to be aware of interdomain links. Therefore, the domains themselves are only responsible for keeping their own information up to date; interdomain links are added manually (cf. Section 4.1).

2.2.1 Domain operations

WSDL Operation Name	**addDomain**
Description	Adds the domain to the NSP
Input parameters (XML type)	Domain : DomainInformationType
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault
Fault (XML type)	DomainAlreadyExistsFault : DomainAlreadyExistsFault

WSDL Operation Name	deleteDomain
Description	Deletes the domain from the NSP
Input parameters (XML type)	DomainIdentifierType : string
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault
Fault (XML type)	DomainNotFoundFault : DomainNotFoundFault

WSDL Operation Name	editDomain
Description	Edits the specified domain link available in the NSP
Input parameters (XML type)	Domain : DomainInformationType
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault
Fault (XML type)	DomainNotFoundFault : DomainNotFoundFault

WSDL Operation Name	getDomains
Description	Retrieves all domains added to the NSP
Input parameters (XML type)	None
Output parameters (XML type)	*Sequence of* Domain : DomainInformationType
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

2.2.2 Endpoint operations

WSDL Operation Name	addEndpoint
Description	Adds an endpoint to the NSP
Input parameters (XML type)	Endpoint : EndpointType

Output parameters (XML type)	Success : Boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault
Fault (XML type)	DomainNotFoundFault : DomainNotFoundFault
Fault (XML type)	EndpointAlreadyExistsFault : EndpointAlreadyExistsFault

WSDL Operation Name	**deleteEndpoint**
Description	Deletes an endpoint from the NSP
Input parameters (XML type)	EndpointID : EndpointIdentifierType
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	**editEndpoint**
Description	Edits the specified endpoint link available in the NSP
Input parameters (XML type)	Endpoint : EndpointType
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	**getEndpoints**
Description	Retrieves all endpoints added to the NSP
Input parameters (XML type)	DomainName : string
Output parameters (XML type)	*Sequence of* Endpoint : EndpointType
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

2.2.3 Interdomain link operations

WSDL Operation Name	addLink
Description	Adds the specified link to the NSP
Input parameters (XML type)	Link : Link
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	deleteLink
Description	Removes the specified link from the NSP
Input parameters (XML type)	LinkId : LinkIdentifierType
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	editLink
Description	Edits the specified link available in the NSP
Input parameters (XML type)	Link : Link
Output parameters (XML type)	Success : boolean
Fault (XML type)	UnexpectedFault : UnexpectedFault
Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

WSDL Operation Name	getLinks
Description	Retrieves all available links in the NSP
Input parameters (XML type)	DomainIdentifierType : string
Output parameters (XML type)	*sequence of* Link : Link
Fault (XML type)	UnexpectedFault : UnexpectedFault

Fault (XML type)	InvalidRequestFault : InvalidRequestFault
Fault (XML type)	OperationNotAllowedFault : OperationNotAllowedFault

2.3 Data types

The following Table 2.1 describes all data types that are used by the reservation and topology interfaces.

Data type name	Description	(composite) Type
ServiceConstraintType	Type used to specify constraints for a service	ServiceIdentifierType : integer TypeOfReservation: ReservationType *one of* FixedReservationConstraints : FixedReservationConstraintType DeferrableReservationConstraints : DeferrableReservationConstraintType MalleableReservationConstraints : MalleableReservationConstraintType AutomaticActivation : boolean *sequence of* Connections : ConnectionConstraintType
ReservationType	Type of reservation	*one of* fixed : string deferrable : string malleable : string
FixedReservationConstraintType	Constraints for fixed reservations	StartTime : dateTime ** Indicates the time when the service should start* Duration : integer ** Indicates the duration of the service in seconds*
DeferrableReservationConstraintType	Constraints for deferrable reservations	StartTime : dateTime ** The earliest point in time when the connection would be useful* Duration : integer ** Indicates the duration of the service in seconds* Deadline : dateTime ** The latest point in time when the service will be useful*
MalleableReservationConstraintType		StartTime : dateTime ** The earliest point in time when the connection would be useful* Deadline : dateTime ** The latest point in time when the service will be useful*

ConnectionConstraintType	Type of constraint on the connection	*extends* ConnectionType MinBW : integer MaxBW : integer MaxDelay : integer DataAmount : long
ConnectionType	Type of the connection	ConnectionIdentifierType : integer Source : EndpointType Target : EndpointType Directionality : integer * *Possible values: 0="unidirectional tree", 1="bidirectional tree", 3="full mesh"*
EndpointType	Information about the endpoint	EndpointID : EndpointIdentifierType Name : string Description : string Interface : EndpointInterfaceType DomainIdentifierType : string Bandwidth : integer * *Bandwidth of the port in Mbps*
EndpointIdentifierType	Type used to identify endpoints	TNAType : string * *Type used for TNA addresses*
EndpointInterfaceType	Interdomain, local end point	*one of* user : string border : string
ConnectionAvailabilityType	Availability of the connection	ServiceIdentifierType : string ConnectionIdentifierType : integer Availability : string * *Allowed values:* (Enumeration) 'available' 'endpoint_not_available' 'path_not_available' 'availability_not_checked' *optional sequence of* BlockedEndpoints : EndpointIdentifierType MaxBW : integer * *Maximum available bandwidth in Mbps (only set if the corresponding MaxBW was set in the availability request)*
ServiceStatusType	Type for the service status	ServiceIdentifierType : string Status : StatusType DomainStatus : DomainStatusType Connections : ConnectionStatusType
StatusType	Type of the status	*one of (Enumeration)* unknown : string pending : string active : string completed : string cancelled_by_user : string cancelled_by_system : string setup_in_progress : string teardown_in_progress : string

DomainStatusType	Type of the domain status	DomainIdentifierType : string Status : StatusType
ConnectionStatusType	Type of the status of the connection	*extends* ConnectionType Status : StatusType DomainStatus : DomainStatusType ActualBW : integer * *Actual bandwidth in Mbps*
GetReservationResponseType	Type for getReservation response	Service : ServiceConstraintType JobIdentifierType : long NotificationConsumerURL : string
GetReservationsComplexType	Type for getReservation response	ReservationIdentifierType : long Reservation : GetReservationResponseType
DomainInformationType	Type for definition of a domain	DomainIdentifierType : string Description : string ReservationEPR : anyURI TopologyEPR : anyURI *sequence of* TNAPrefixType : string
Link	Type for definition of a link	*extends* LinkIdentifierType Name : string Description : string Delay : integer
LinkIdentifierType	Identifies a link	SourceEndpoint : EndpointIdentifierType DestinationEndpoint : EndpointIdentifierType
UnexpectedFault	Returned in case an internal error occurred and contains information describing the fault encountered	*extends* wsbf:BaseFaultType * This type is described in OASIS Web Services Resource Framework (http://www.oasis-open.org/specs/index.php#wsrfv1.2)
InvalidRequestFault	Returned in case that the request doesn't match the interface specification	*extends* wsbf:BaseFaultType * This type is described in OASIS Web Services Resource Framework (http://www.oasis-open.org/specs/index.php#wsrfv1.2)
OperationNotAllowedFault	Returned in case that the operation is not allowed with the given user credentials	*extends* wsbf:BaseFaultType * This type is described in OASIS Web Services Resource Framework (http://www.oasis-open.org/specs/index.php#wsrfv1.2)
EndpointNotFoundFault	Returned in case a given endpoint	*extends* ReservationFault

	cannot be located	
ReservationFault	Base fault type for reservation interface operation faults	*extends* wsbf:BaseFaultType * This type is described in OASIS Web Services Resource Framework (http://www.oasis-open.org/specs/index.php#wsrfv1.2)
DomainAlreadyExistsFault	Returned in case the domain already exists	*extends* TopologyFault
DomainNotFoundFault	Returned in case the domain can not be recovered	*extends* TopologyFault
EndpointAlreadyExistsFault	Returned in case that the endpoint already exists	*extends* TopologyFault
TopologyFault	Base fault type for topology interface operation faults	*extends* wsbf:BaseFaultType * This type is described in OASIS Web Services Resource Framework (http://www.oasis-open.org/specs/index.php#wsrfv1.2)

Table 2.1: Data types used on the northbound interface

3 Network Service Plane architecture

3.1 Architecture overview

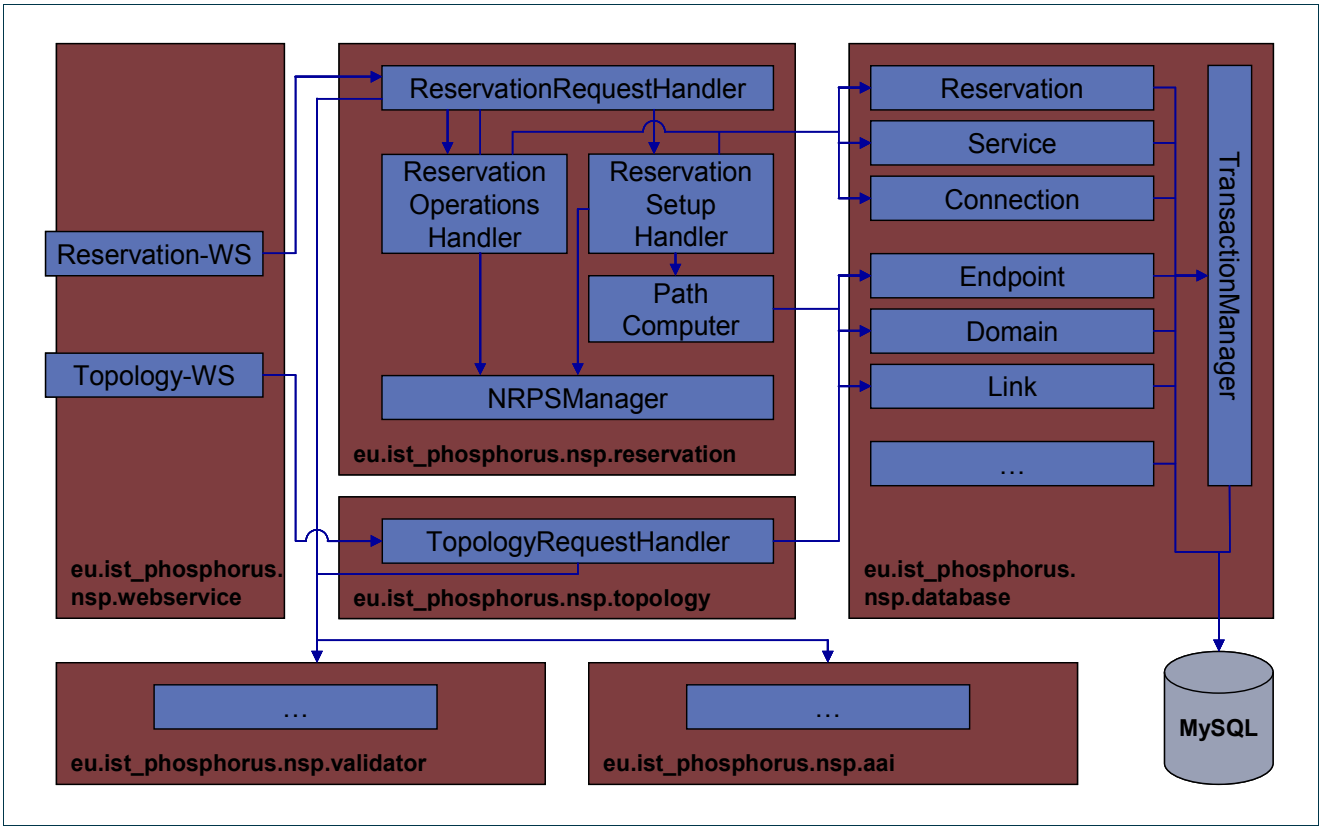

Figure 3.1: NSP architecture (simplified)

Figure 3.1 depicts a simplified overview of the NSP architecture and the dependencies between the different modules. Requests from the middleware or from a user application are received through the webservice classes in the package *eu.ist_phosphorus.nsp.webservice*. These auto-generated classes contain very little functionality and mainly hand the received requests to a *CommonRequestHandler* class (not depicted in the figure) that validates the requests using the *eu.ist_phosphorus.nsp.validator* package (cf. Section 3.2.3), that authorize the requests using the *eu.ist_phosphorus.nsp.aai* package (cf. Section 3.3), and that finally forwards the requests to the appropriate handlers.

The Java classes that represent the data communicated through the webservice interfaces are auto-generated from the WSDL specifications using JAXB (Java Architecture for XML Binding, [JAXB], cf. Section 3.2.1). Data to be stored persistently is represented through own Java classes in the *eu.ist_phosphorus.nsp.database*

package (cf. Section 3.4). These classes offer all functionality for database access, so it is not necessary that classes outside of this package are aware of details of the database used.

Requests received through the Topology-WS are processed by the *TopologyRequestHandler* (cf. Section 3.1.1), whereas requests received through the Reservation-WS are processed by the *ReservationRequestHandler* (cf. Section 3.1.2). This possibly results in communication between NSP modules and different NRPS adapters, which is encapsulated by the *NRPSManager* class (cf. 3.6).

3.1.1 Topology webservice

All requests received through the Topology-WS are processed by the *TopologyRequestHandler* class. Basically, this class "translates" between the JAXB classes used on the webservice interface and the database classes used in the NSP core and vice versa.

3.1.2 Reservation webservice

Requests received through the Reservation-WS are processed by the *ReservationRequestHandler* class. This class serves as a facade towards the *eu.ist_phosphorus.nsp.reservation* package; its sole purpose is to distribute requests to the different handler classes responsible for specific tasks.

The *ReservationSetupHandler* processes requests that are related to the establishment of new reservations (cf. Section 2.1.2). This class checks the availability of network resources and sets up reservations. Both operations interact with the path computation module (cf. Section 3.5) and share some other functionality.

This is not the case for the other operations that are implemented in the *ReservationOperationsHandler*, which processes requests that are related to previously established connections. It retrieves reservation data, checks the status of reservations, cancels reservations and activates previously reserved services (in case the auto activation feature was not requested).

3.2 Request and exception handling

The webservice/SOAP request, response and exception handling is abstracted in WP1 by using a unified request handling layer. This layer is used in the NSP and each NRPS adapter. Its task is to create a transparent communication between Java based clients (e.g. Meta-Scheduling Service, NSP, NRPS adapter) without dealing with webservice related issues. As shown in Figure 3.1 each webservice forwards the requests to its corresponding request handler. This handler cares about the data binding, exceptions and the request validation.

3.2.1 Data binding

In order to abstract the Java based system from any XML related functions, the Java Architecture for XML Binding (JAXB) [JAXB] is used. Incoming XML requests are transparently unmarshalled into annotated Java objects that can easily be used to retrieve or modify the submitted data. The Java response objects then are marshalled back into the according XML representation. This way, JAXB allows storing and retrieving data in memory in any XML format, without the need to implement a specific set of XML loading and saving routines for the program's class structure. The needed JAXB Java objects are generated automatically by using the XML Schema of the webservice request and response types. By using this construct also exceptions can be exchanged through the webservice.

3.2.2 Exception handling

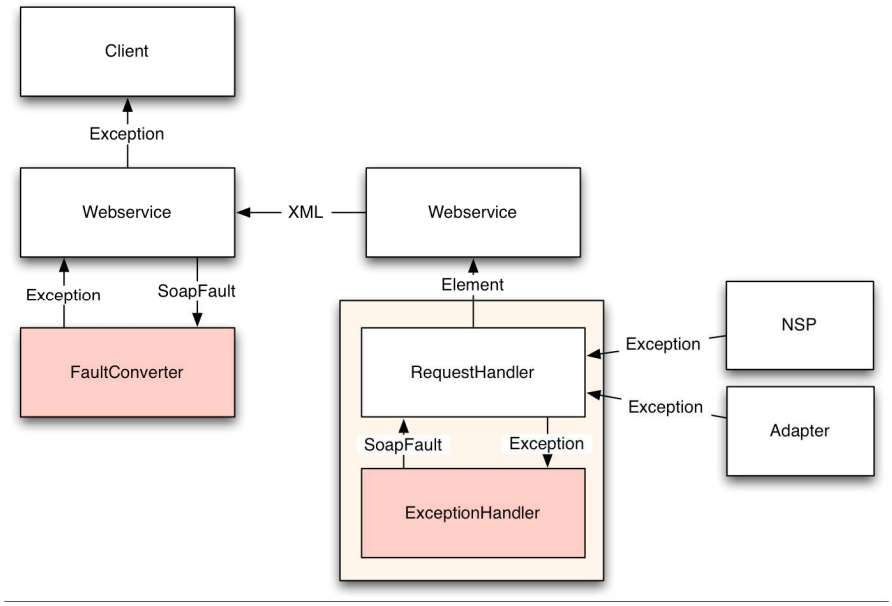

Figure 3.2: Exception handling

Exceptions are a contruct of the Java programming language to handle expected or unexpected failures. A webservice has a similar approach and it uses so called SoapFaults to communicate failures.

As shown in Figure 3.2, if an exception is thrown by the NSP or an NRPS adapter, it is caught by the appropriate request handler and forwarded to the exception handler. The exception handler converts the Java exception including the message and the stacktrace to an XML stream. This stream, which contains the complete Java stackstrace in the SoapFault “Detail” field, is sent out to the client via the webservice as a predefined SoapFault (cf. Section 2). On the client side, the SoapFault is converted back to the original Java

exception by the client's webservice. This way the NSP and the NRPS adapters can handle failures efficiently and communicate all the information to the client. From the client point of view, the called webservice operation acts as a normal Java method.

3.2.3 Validation

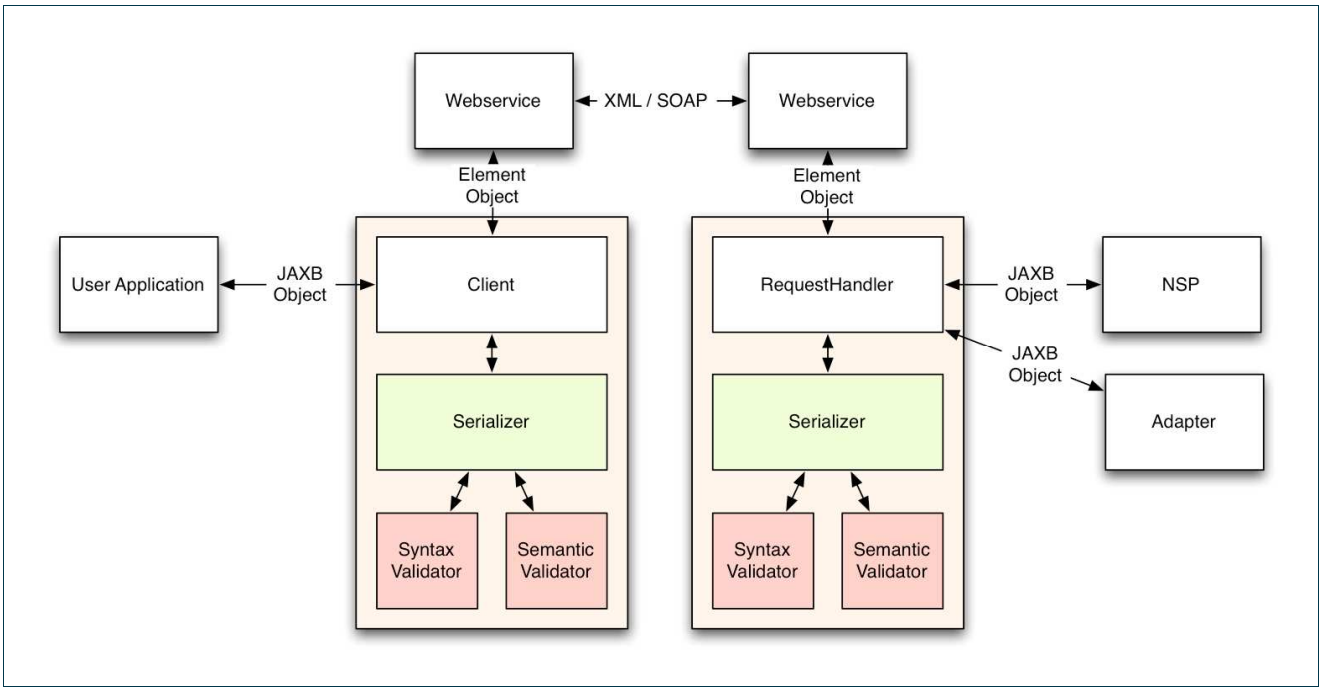

Figure 3.3: Request validation

Before sending or forwarding any (XML) data, the requests and responses are validated in the NSP and the NRPS adapters. As shown in Figure 3.3, the incoming and outgoing messages are validated in two ways. First, the structure itself is validated by the syntax validator. The message must conform to the XML Schema definition of the corresponding type. If this is not the case, a syntax validation exception is thrown and the message is discarded. If the syntax is correct, the semantic validator verifies constraints regarding the content. If e.g. the requested minimum bandwidth is higher than the requested maximum bandwidth in a message, a semantic validation exception is thrown and again the message is discarded. This way, the core system can assume to retrieve only valid data.

3.3 Authentication and authorization

As shown in Figure 3.4, the NSP contains a central module that is used to authenticate all incoming and to sign all outgoing traffic. This Message Level Security (MLS) service is based on the OASIS Webservices Security standard [WSS]. Besides procedures to sign and to encrypt SOAP messages the standard includes options to

attach security credentials like username/password, X.509 certificates or tokens. The NSP itself currently does not provide complex authorization or accounting mechanisms. Rather it forwards user credentials (attributes) that are contained in the incoming message from the middleware to the involved NRPS adapters and vice versa. The authorization process is in the scope of each NRPS and the middleware – authorization tickets are transparently communicated through the NSP. It is assumed that each domain has its own policy and attribute database. The NRPS adapter may map the global attributes to local ones or local user accounts. In the case that global and local attributes are identical, this mapping reduces to the identity function.

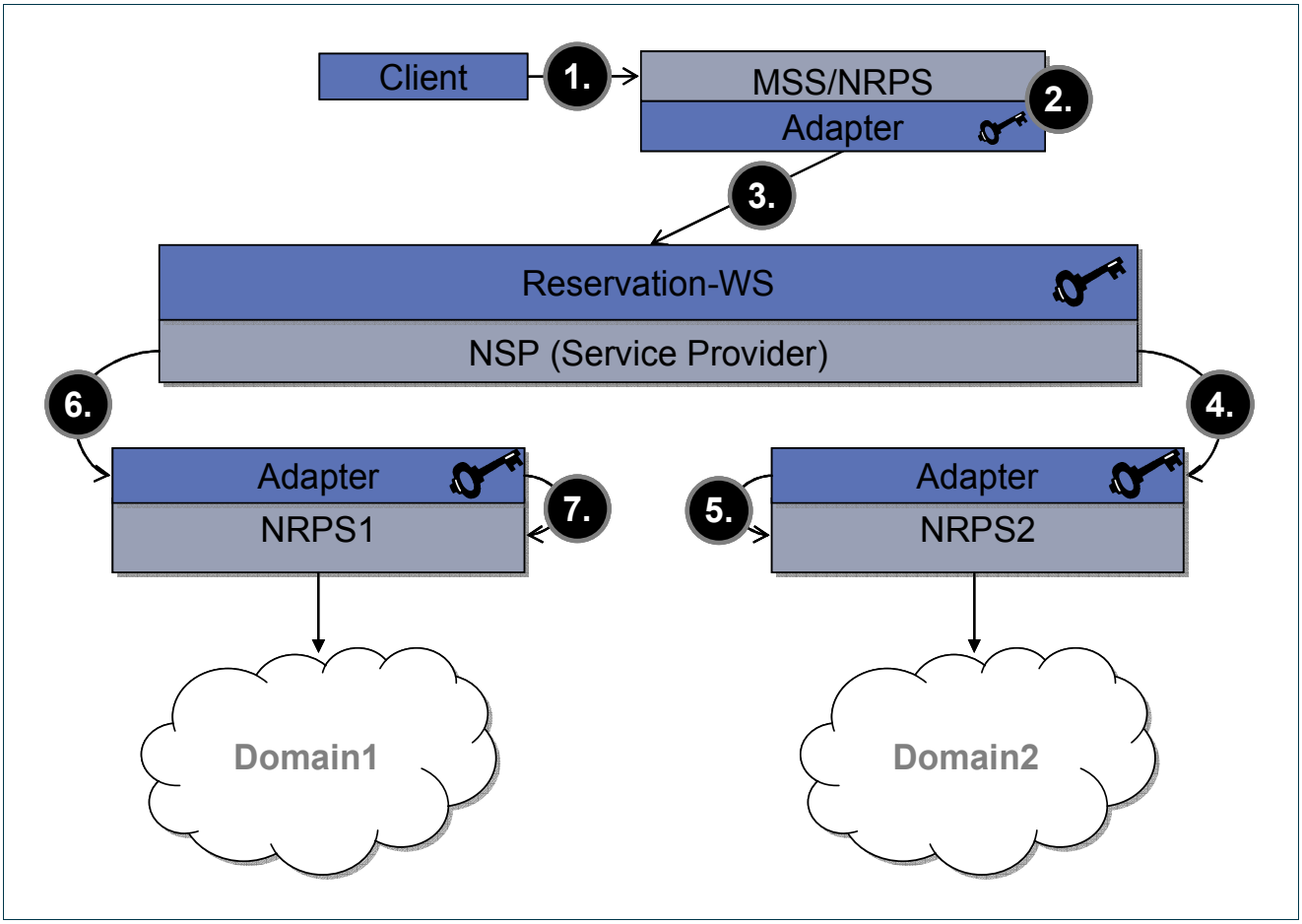

Figure 3.4: Authenticated message flow

Since the communication with the NSP requires valid authentication credentials, the NSP creates a transitive trust relation between all participating partners. This is achieved by preinstalling the NSP public key in the middleware and in each NRPS adapter. On the other hand, the NSP must possess the public keys of each communication party in advance.

Figure 3.4 depicts a sequence of interactions needed for the authentication flow mentioned above. The following sequence description is simplified and reduced to a single MSS-NRPS communication. The initial client request could also be sent to an NRPS.

(1) A client sends a request to the Meta-Scheduling Service (MSS) with local user credentials. (2) The MSS authorizes the user and the request locally. In case the request is authorized successfully, the scheduler maps the user credentials to accordant global attributes, adds these to the request to the NSP and signs the message with its private key. (3) The message then is sent to the NSP on behalf of the client. Since the public key of the MSS is trusted within the NSP, the message is accepted in the next step. The signature of the valid incoming request will be removed and the request may be split into several new requests. (4)(6) The outgoing messages to the NRPS adapters are signed by the NSP. All authorization related information that may be added by the MSS is forwarded without any modification. In the expected case that the NRPS adapter trusts the NSP key all authorization information (e.g. global attributes, tickets) and the request are forwarded to the specific NRPS. (5)(7) A complex authorization process has to be implemented in the NRPS adapter or the NRPS itself.

This way, the NSP acts as a transparent broker between the middleware and each NRPS. It is self-evident that this message level security flow is also applied for the corresponding response messages. Additionally, this architecture could be used to encrypt the whole message flow.

3.4 Data storage

3.4.1 Storage structure

This section describes the Entity Relationship (ER) model used in the database of a NSP instance. See Figure 3.5 for an overview. There are entities related to reservation management and entities related to topology management. The entities are described in detail in the following two subsections.

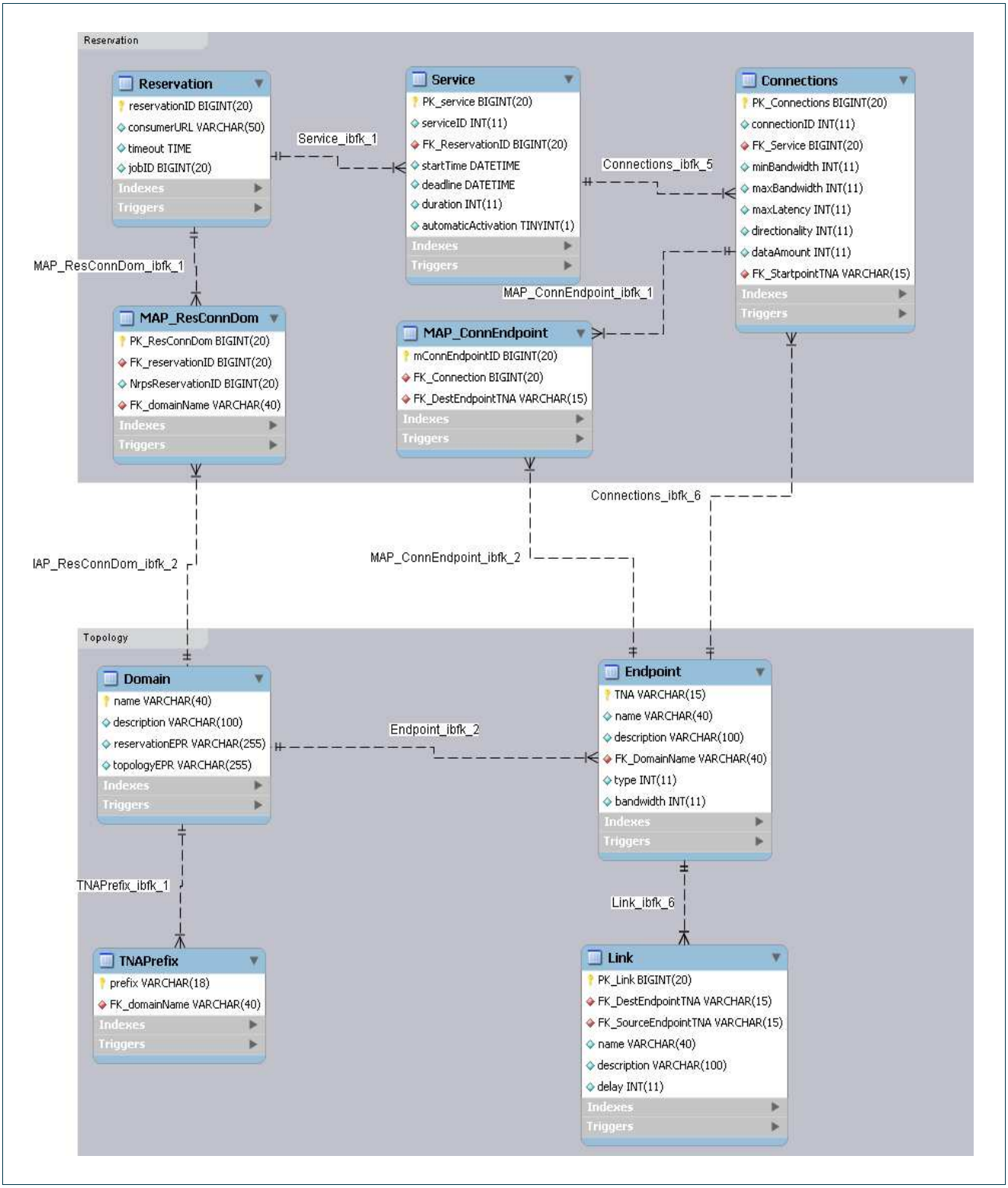

Figure 3.5: Entity Relationship model of the NSP database

3.4.1.1 *Reservation management*

The data types used in the Reservation-WS have been introduced in D1.1. Recall that a *reservation* is composed of one or more *services*, and that a *service* in turn is composed of one or more *connections*. A *service* actually specifies the type of the requested service and introduces constraints in the time domain, whereas a *connection* introduces topological constraints. The following tables are used to store this data.

Table: *Reservation*		
Column Name	**Column Type**	**Description**
reservationID	bigint(20)	**Primary key.** Auto-generated reservation ID, unique within the NSP.
consumerURL	varchar(255)	Optional URL of Notification-WS.
timeout	datetime	Timeout for pre-reservations. For a permanent reservation, this value is not set.
jobID	bigint(20)	ID of the job this reservation is related to.

Table: *Service*		
Column Name	**Column Type**	**Description**
PK_service	bigint(20)	**Primary key.** Auto-generated unique ID, only used in the database context.
serviceID	int(11)	Service ID specified by the user, makes service unique within a single reservation.
FK_ReservationID	bigint(20)	**Foreign key.** Identifies the reservation this service is associated with.
startTime	datetime	Starting time of this service (earliest start time in case of deferrable or malleable reservations).
deadline	datetime	Deadline for malleable reservations. Not set for fixed or deferrable reservations.
duration	int(11)	Duration of a fixed or deferrable service. Not set for malleable reservations.
automaticActivation	tinyInt(1)	Boolean value indicating whether the service is to be activated automatically or by an activate operation.

Table: *Connection*		
Column Name	**Column Type**	**Description**
PK_Connection	bigint(20)	**Primary key.** Auto-generated unique ID, only used in the database context.
connectionID	int(11)	Connection ID specified by the user, makes connection unique within its enclosing service.
FK_Service	bigint(20)	**Foreign key.** Identifies the service this connection is associated with.
minBandwidth	int(11)	Minimum bandwidth of the connection (in Mbps).
maxBandwidth	int(11)	Optional maximum bandwidth of the connection (in Mbps).
maxLatency	int(11)	Optional maximum latency of the connection (in milliseconds).
directionality	int(11)	Directionality of the connection: There are numerical values for “unidirectional”, “bidirectional tree”, and “full mesh”.
dataAmount	int(11)	Optional data amount to be transferred (only used with malleable reservations).
FK_StartPointTNA	varchar(15)	**Foreign key.** Source endpoint of this connection.

Note that the NSP is designed to support point-to-multipoint connections, therefore a separate table called *MAP_ConnEndpoint* is required to store the target(s) of a connection.

Table: *MAP_ConnEndpoint*		
Column Name	**Column Type**	**Description**
mConnEndpointID	bigint(20)	**Primary key.** Auto-generated unique ID, only used in the database context.
FK_Connection	bigint(20)	**Foreign key.** Identifies the connection this entry is associated with.
FK_DestEndpointTNA	varchar(15)	**Foreign key.** Identifies the endpoint that is a target of the connection.

The mapping between Reservation IDs allocated by the NSP and Reservation IDs allocated by the NRPSs is stored in the *MAP_NRPSResvID* table.

Table: *MAP_NRPSResvID*		
Column Name	**Column Type**	**Description**
PK_NRPSResvID	bigint(20)	**Primary key.** Auto-generated unique ID, only used in the database

		context.
FK_reservationID	bigint(20)	**Foreign key.** Identifies the reservation this entry is associated with.
NrpsReservationID	bigint(20)	ID allocated to this reservation by the NRPS of an underlying domain.
FK_domainName	varchar(15)	**Foreign key.** Identifies the domain whose NRPS allocated the reservation ID.

3.4.1.2 *Topology management*

Table: *Domain*		
Column Name	**Column Type**	**Description**
name	varchar(40)	**Primary key.** Domain name.
description	varchar(100)	Optional descriptive text for a domain.
reservationEPR	varchar(255)	Reservation-WS for accessing the domain's NRPS.
topologyEPR	varchar(255)	Optional Topology-WS for administrative access to the domain's topology.

Table: *Endpoint*		
Column Name	**Column Type**	**Description**
TNA	varchar(15)	**Primary key.** Endpoint's TNA in IPv4 syntax, stored as string.
name	varchar(40)	Optional endpoint name.
description	varchar(100)	Optional endpoint description.
FK_DomainName	varchar(40)	**Foreign key.** Identifies the domain this endpoint is located in.
type	int(11)	Numerical value that discriminates between user and border endpoint.
bandwidth	int(11)	Bandwidth of the endpoint (in Mbps).

Table: *TNAPrefix*		
Column Name	**Column Type**	**Description**
prefix	varchar(18)	**Primary key.** Prefix in IPv4 notation, e.g. "10.1.3.0/24".
FK_domainName	varchar(40)	**Foreign key.** Identifies the domain this prefix is associated with.

Table: *Link*		
Column Name	**Column Type**	**Description**
PK_Link	bigint(20)	**Primary key.** Auto-generated unique ID, only used in the database context.
FK_DestEndpointTNA	varchar(15)	**Foreign key.** Identifies the link's destination endpoint.
FK_SourceEndpointTNA	varchar(15)	**Foreign key.** Identifies the link's source endpoint.
name	varchar(40)	Optional endpoint name.
description	varchar(100)	Optional endpoint description.
delay	int(11)	Optional delay of the link.

3.4.2 Data binding

From the Java code of the NSP, the database is accessed using the Hibernate software [Hibernate]. Each of the entities introduced in the previous section is represented by a Java class that encapsulates all functionality necessary for database access. This means that outside of these classes, there is no need to be aware of details specific to the database.

These Java classes are different from the classes used to represent objects transported across the webservice interfaces; e.g., the *Domain* class used internally to represent a domain differs from the *DomainType* used on the Topology-WS. Therefore, the NSP can easily be extended by other interfaces without having to change the code at the system core.

3.5 Path computing

The Path Computer module is designed to calculate interdomain paths using the Dijkstra algorithm. The interfaces of the Path Computer are shown in the following tables. When a new instance of the Path Computer is created, it reads all border endpoints of all domains and all interdomain links from the NSP database. After that, one ore more services together with their beginning and ending times are added. Path computation requests are grouped per service. When calculating a path, blocking of resources (i.e. resources that are in use by another service that is at least partly overlapping in time) is taken into account.

This workflow is shown in Figure 3.6. All paths belonging to a specific service are calculated at the same time but can be selectively read from the Path Computer one by one. Each path is returned as a list of tuples of endpoints. Each tuple consists of two endpoints of the same domain. The calculation of the intradomain parts of

a path is left to the Path Computer of the domains' NRPSs. If an NRPS returns that one or both endpoints are not available, or no connection between the endpoints is possible for this request, the unavailable resources can be pruned for this path computation and a new set of paths can be computed. As the Path Computer is able to calculate shortest paths after pruning of previously selected endpoints and/or intradomain paths, it is not required to know in advance if endpoints or connections are available.

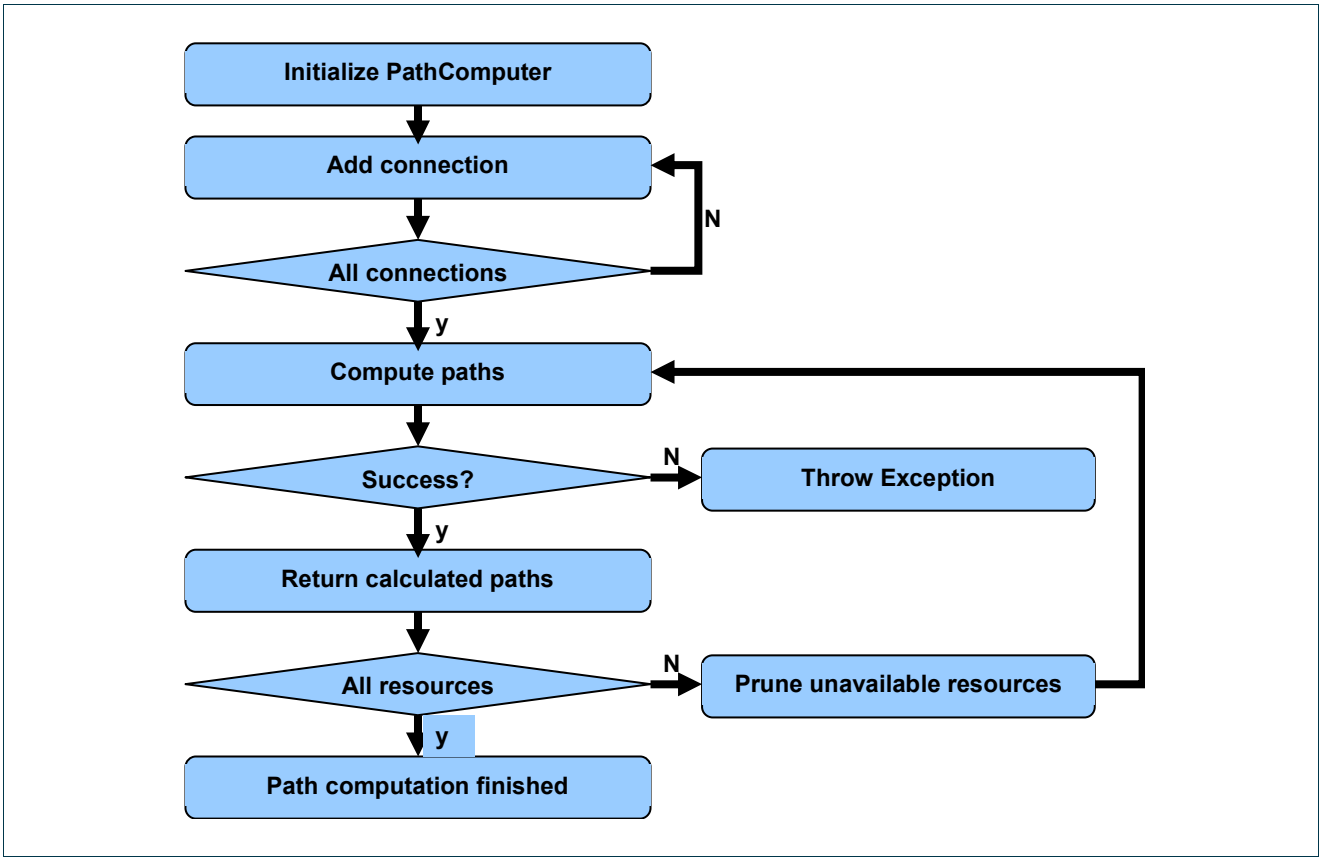

Figure 3.6: Path Computer Workflow

Interface	addService
Parameters	startTime : long endTime : long serviceId : int
Result type	void
Exceptions	InvalidServiceIdException
Description	Add a service to the path computer's state. The start and end times can be given in arbitrary time units since they are only used to calculate which services overlap in time and which do not.

Interface	addConnection
Parameters	source : Endpoint destination : Endpoint serviceId : int connectionId : int
Result type	void
Exceptions	EndpointNotFoundFaultException DatabaseException InvalidServiceIdException InvalidConnectionIdException
Description	For a specific service, add a connection from a source to a destination to the path computer's state.

Interface	computePaths
Parameters	serviceId : int
Result type	void
Exceptions	PathNotFoundFaultException InvalidServiceIdException
Description	Compute paths for all connections that belong to a specific service.

Interface	getPath
Parameters	serviceId : int connectionId : int
Result type	List<Tuple<Endpoint, Endpoint>>
Exceptions	PathNotFoundFaultException InvalidServiceIdException
Description	Retrieve shortest path for a certain connection. Requires that the path was already computed using computePaths.

Interface	pruneEdge
Parameters	serviceId : int connectionId : int src : Endpoint

	dst : Endpoint
Result type	void
Exceptions	EndpointNotFoundFaultException InvalidServiceIdException InvalidConnectionIdException
Description	Prune an intradomain edge from the internal topology graph of this path computer instance.

Interface	pruneEndpoint
Parameters	serviceId : int connectionId : int endpoint : Endpoint
Result type	void
Exceptions	EndpointNotFoundFaultException InvalidServiceIdException InvalidConnectionIdException
Description	Prune an endpoint from the internal topology graph of this path computer instance.

3.6 Request forwarding

When coordinating multidomain reservations, the NSP typically has to forward multiple requests related to a single multidomain reservation to different NRPSs.

The module of the NSP in charge of the communication with the NRPSs is the NRPS Manager. This module is invoked internally by other blocks of the NSP when one or more messages have to be sent to one or more NRPSs. When the manager is invoked, it creates a proxy for each NRPS webservice, sends the message and waits for the responses. After, it returns the responses to the invoker.

The responses received in reply to these requests have to be analysed to take further action or to construct a response in reply to a request received on the reservation interface of the NSP. However, some types of replies require immediate action. If a single request in a series of *CreateReservation* requests fails, then a rollback is required. Hence, the reservations that have already been established in other domains must be cancelled.

The requests to the NRPS manager are composed by tuples <Domain, MessageToSend>, and the responses consist of pairs <Domain, ResponseReceivedFromNRPSAdapter>. All these responses coming from each NRPS adapter are returned to the invoking request handler. In the case that the overall reservation process has

not been successful (i.e. error returned by one or more NRPSs), the NRPS manager will create the necessary *CancelReservation* requests to domains that have reported a successful reservation creation.

The NSP and an NRPS communicate through the Reservation-WS offered by each NRPS. This interface implements the same operations as the northbound reservation interface of the NSP. The NSP must be aware of the endpoint reference of each Reservation-WS in order to be able to invoke its operations. This information is stored in the database together with the other required information about the domain controlled by the NRPS.

For the project, two versions of the NRPS manager have been implemented. The first one is based on a sequential version (chain model) whereas the second one is a concurrent version (threaded model). The sequential manager is the first implementation step for a prototype; but lacks of scalability (even in a single level hierarchy) and may cause high delays in the reservation process, as it will be explained in the coming paragraphs. Therefore, this sequential manager will be replaced by the threaded model, which means a second implementation step for an improved, scalable prototype. Moreover, the threaded manager will facilitate deploying multi-level hierarchies of NSP entities in the future.

The sequential NRPS manager sends a request to an NRPS, waits for the response and then continues with the next NRPS, until the last NRPS has responded. This model can cause large delays in the NSP, especially with a larger amount of NRPSs involved, since there is only one NRPS working at a time and the NSP has to wait for the response of each one separately. The overall response time in the request forwarding process will be the addition of the response times of all the NRPSs, so this first prototype does not scale well for a large number of NRPSs, due to the linear growth in average of the overall response time caused by the number of NRPSs. The following equation represents the total response time of the system, *T*, where*N* is the number of NRPSs involved in the provisioning request and t_i the response time of an NRPS:

$$T = \sum_{i=1}^{N} t_i$$

It must be taken into account that t_i is strongly dependant on the size of the transport network to be configured by the NRPS and the response time of the network hardware.Figure 3.7 depicts the operation workflow for the sequential NRPS Manager. The numbered arrows show the order in which the requests are sent to each NRPS Adapter.

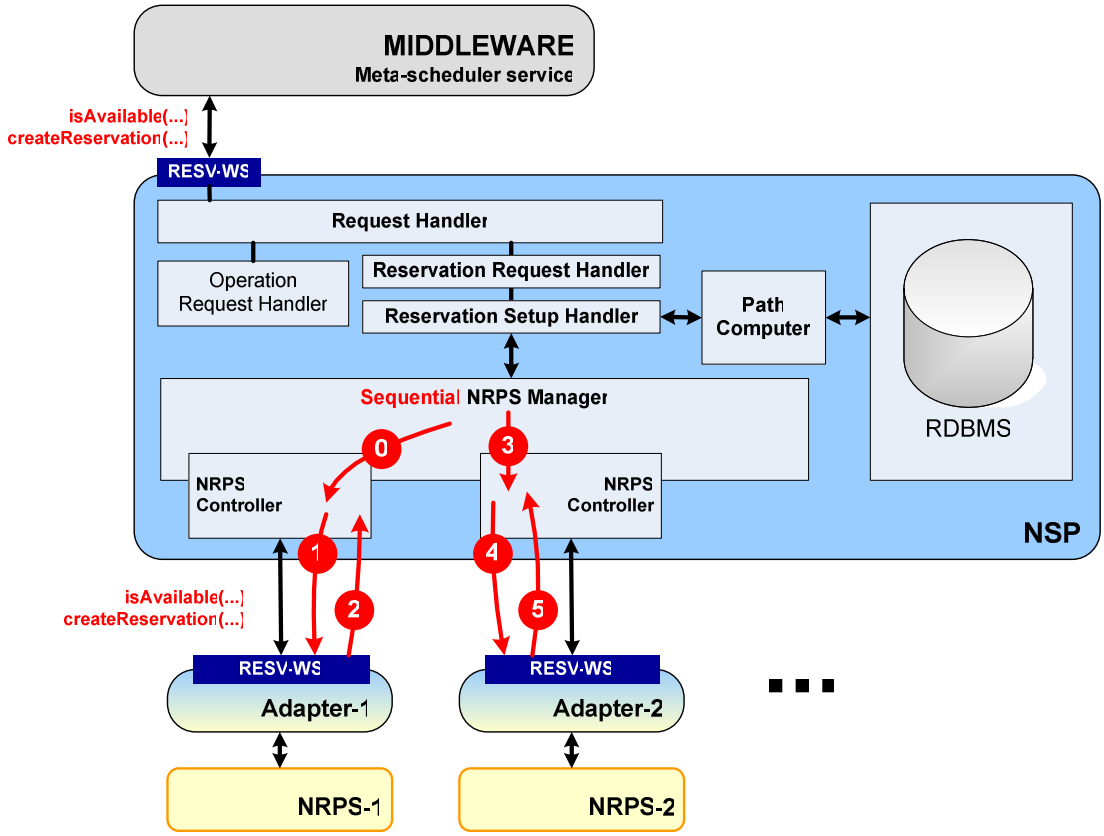

Figure 3.7: Sequential NRPS Manager (1st prototype).

The second version is threaded, developed with the idea to avoid large waiting times by sending the requests to all the NRPSs in parallel. This way, all the NRPSs receive the request nearly at the same time and they can work concurrently, shortening the overall response time in the request forwarding process. This model will have as an overall response time the longest response time of all the NRPSs, so there is no dependency on the number of NRPSs, *N*, but on the maximum response time of all involved NRPSs. This allows the system being scalable enough by making *T* a convergent function of the local time response of an NRPS. This is shown in the following equation:

$$T = \max_i t_i$$

Moreover, this threaded model of the second NRPS manager will facilitate and shorten in time the rollback periods in case of reservation process failures. As commented before, each NRPS controller thread communicates with a single NRPS. If a single reservation query to an NRPS fails, the corresponding NRPS controller will report the problem (failure message / fault) to the NRPS manager and it will cancel all reservations or pre-reservations done for this E2E connection (or job). Hence, the threaded manager will send a *CancelReservation* message to each controller, in parallel, so the cancel process will be done concurrently,

as the creation process was. Moreover, the overall response time for the cancel reservation process follows the equation above, since the NRPS manager has the same behaviour for both requests.

Figure 3.8 depicts the operation workflow for the threaded NRPS Manager. Again, the numbered arrows show the order in which the requests are sent to each NRPS Adapter.

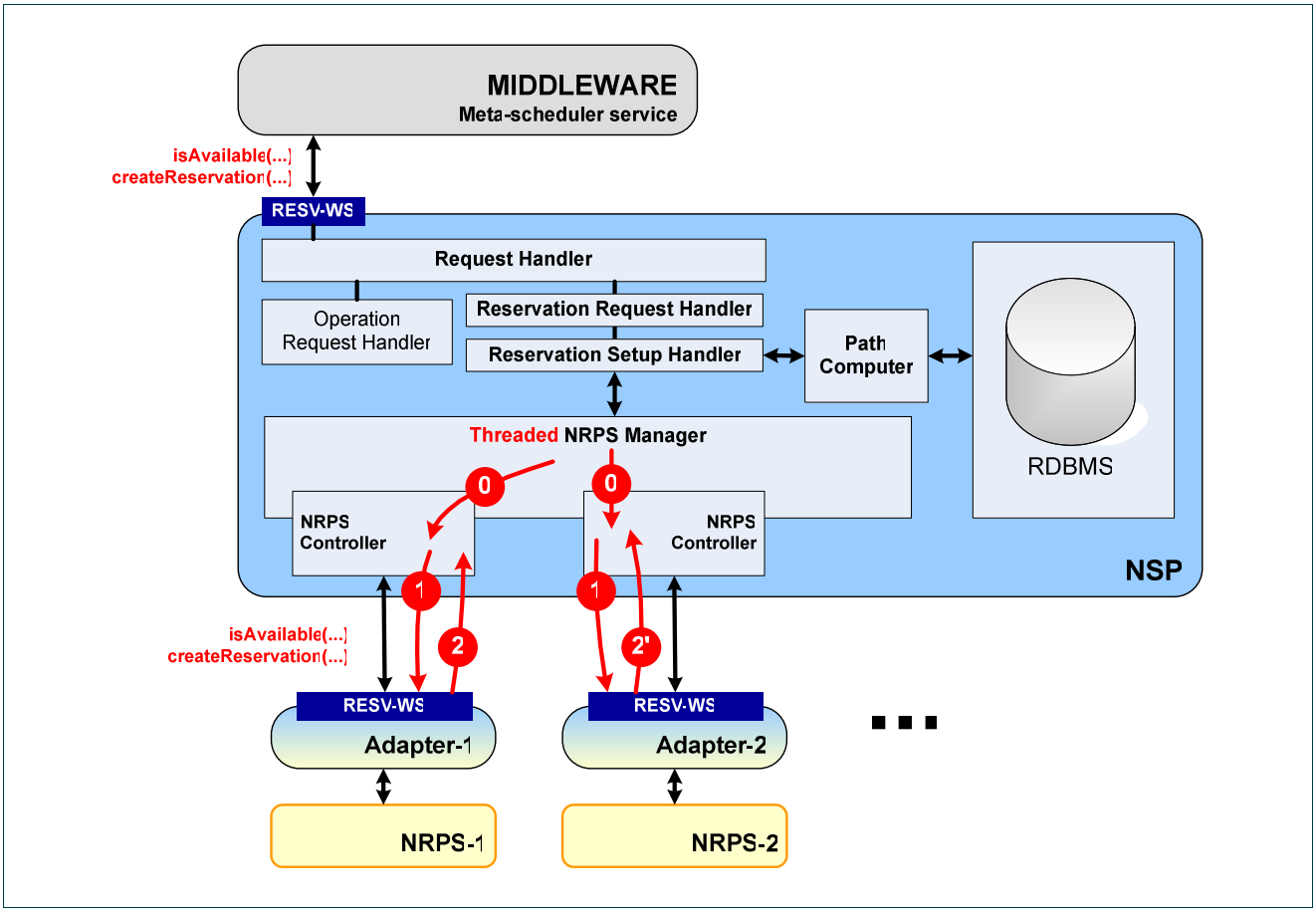

Figure 3.8: Threaded NRPS Manager (2nd prototype).

Finally, it must be taken into account the possibility of using multi-level hierarchical architectures of different NSPs, i.e. placing one or more NSPs under another NSP at various levels, coexisting with NRPSs at the different levels. The study of these hybrid centralized/distributed architectures is out of the scope of this document, but must be considered because it could enhance the scalability of the overall system and even improve the communication/information exchange with other systems considered in the second phase of the Phosphorus project.

4 Network Service Plane functionality

4.1 Topology modification

The topological information about the network that is stored in the database is managed through the Topology-WS interface of the NSP. This interface can be accessed by any client. For a human user, the Topology Client, a graphical user interface (GUI), has been developed.

The Topology Client allows the user to create, query, modify and delete any topological information of the network (add, modify, query and delete domains, endpoints and interdomain links). The GUI is implemented in Java Swing and contains a proxy to communicate with the Topology-WS of the NSP.

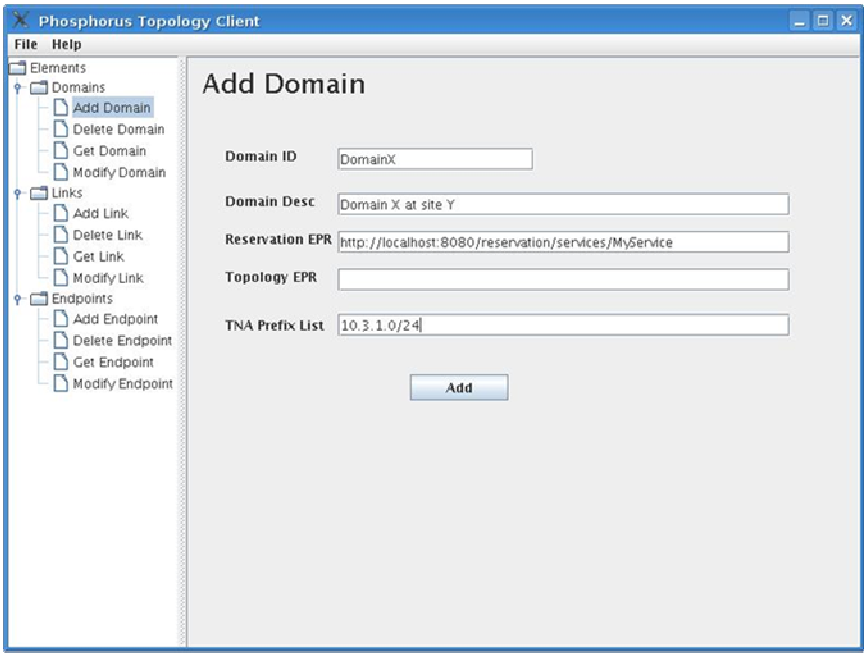

Figure 4.1: Topology Client GUI

The natural way to add the domains and endpoints is not through the GUI (although the user can do it this way as well), but by means of an automatic registration process located at the NRPS Adapter. When the adapter is initialized, a servlet is executed that contacts the NSP in order to send to it the information about the domain (identifier, description, WS endpoint references, ...). Once the adapter has registered, the NSP has the information required to send requests to the NRPS. After the registration, the adapter starts a process that updates the NSP with the endpoints of the domain controlled by the adapter. This way, the NSP is updated periodically with the information of the local topology of each one of the NRPSs.

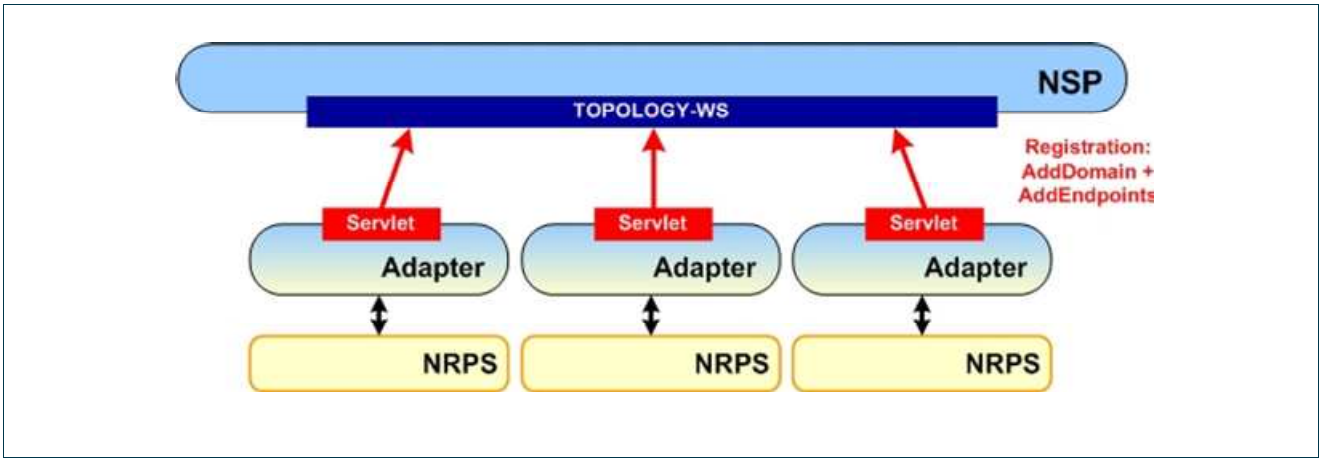

Figure 4.2: Domains registration

The way to indicate the interdomain links is not automatic and must currently be done manually through the Topology Client, since the NRPSs do not have this interconnectivity information internally. In this case, the user has to select the endpoints of the link manually and insert some related information to add the link to the topological database.

4.2 Network resource availability query

An availability query is triggered by the reception of an *IsAvailable* request by the Reservation-WS.

Upon reception of an *IsAvailable* request, the corresponding routine in the *ReservationSetupHandler* class is called. Though a reservation will actually not be made, the task to be solved is very similar to an actual reservation, the only exception being that the queried resources are not reserved. Also, alternative start time offsets can be returned, while an actual reservation only either succeeds or fails.

The requested services are used as input for the *getAvailableServiceList* routine that is also used for *CreateReservation* requests. This routine queries the *PathComputer* for paths for all of the requested connections and splits the single multidomain request to multiple single domain requests, one for each of the involved domains. These requests are handed to the *NRPSManager* that takes care of sending these requests to the NRPSs and collecting the corresponding replies.

If the requested resources are not available in one or more of the involved domains, they are pruned from the *PathComputer* instance (cf. Section 3.5). In this case, the domains should reply with alternative start time offsets, the latest of which is recorded for later use if no suitable path can be found. Now, the *PathComputer* is queried again for an alternative path.

This process is repeated until either a suitable path is found, or until so many resources have been pruned that no path is available for one or more connections. In the first case, the requestor is informed that the resources

are available. In the latter case, the requestor is informed that they are not available, and the earliest alternative start time offset of those recorded as described above is reported as an alternative start time offset.

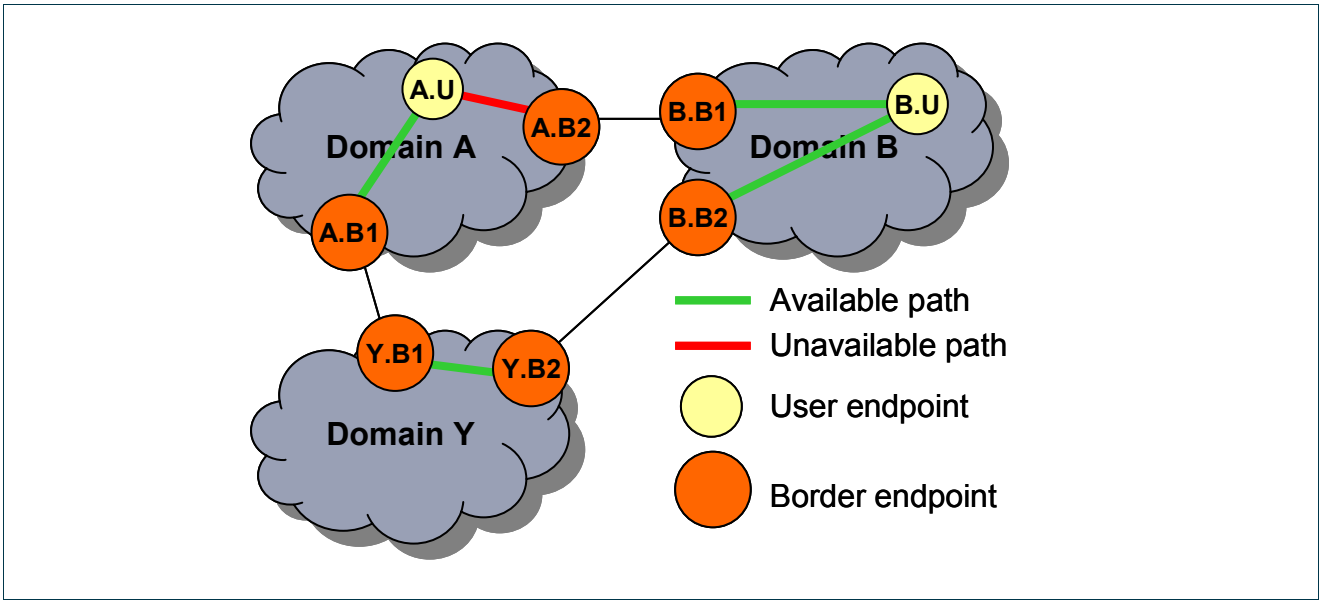

Figure 4.3: Example scenario for reservation setup

To illustrate this, consider the scenario sketched in Figure 4.3. The NSP is processing an *IsAvailable* message for a connection between A.U and B.U. The path computation first returns the two intradomain connections A.U-A.B2 and B.B1-B.U. *IsAvailable* messages for these partial paths are then forwarded to domains A and B. A replies that A.U-A.B2 is not available, while B replies that B.B1-B.U is available. Therefore, the intradomain connection A.U-A.B2 is pruned from the *PathComputer* instance handling this request. If the path computer would not find an alternative path between A.U and B.U, then the NSP would reply with a negative *IsAvailableResponse*; an alternative start time offset returned by A would be included in the NSP's reply.

In this example however, there is an alternative path, so the path computation would yield three intradomain connections A.U-A.B1, Y.B1-Y.B2, and B.B2-B.U in the next iteration. All three corresponding *IsAvailable* requests to the domains A, B, and Y are answered positively. Thus, also the NSP's reply to the *IsAvailable* request for the interdomain path A.U-B.U is positive.

4.3 Network resource reservation

The reservation of network resources is internally handled similar to the availability query described in the previous section. Before sending *CreateReservation* messages to the NRPSs, the availability of the requested resources is checked. This is to prevent a series of *CreateReservation* and *CancelReservation* messages that would be necessary if one or more domains in a multidomain path are not able to fulfill the reservation.

Alternative start times reported by the NRPS adapters are however discarded. The availability of intradomain paths along different routes returned by the *PathComputer* is checked merely with the constraints specified in the *CreateReservation* message. In case a path consisting of domains that all gave a positive reply to the availability query is found, the final reservations for all intradomain paths are established.

Considering the exampled sketched in Figure 4.3 again, upon a *CreateReservation* request for the connection A.U-B.U, the NSP would first check for an available path just as described in the previous section. Only then will the three *CreateReservation* requests for the intradomain paths A.U-A.B1, Y.B1-Y.B2, and B.B2-B.U be sent to the corresponding domains.

4.4 Reservation status query

A *GetStatus* is mapped to a set of single-domain *GetStatus* queries in a straightforward way. From the reservation ID that is part of this message, the domains and the reservation IDs used for this reservation inside each of the domains are retrieved from the database, and a set of corresponding *GetStatus* messages is constructed and passed to the *NRPSManager*.

Practical considerations during first testing of the code have lead to a slight modification of the *GetStatusResponse* messages. In addition to an overall status code for each connection that is generated from the set of status codes for this connection received from the participating NRPSs, the *GetStatusResponseType* optionally contains *DomainStatus* elements, each of which contains a domain name and an element of type *ConnectionStatusType*, i.e. the connection status received from the specified domain. This is mainly interesting for debugging purposes in cases where the status values are not consistent. E.g., in case a connection should be established, all domains should return the status code *active*. If one domain returns a different status code, it is immediately visible in which domain the error has occurred.

4.5 Reservation cancellation / connection teardown

An already established reservation is cancelled by a *CancelReservation* message. For the NSP, it is not of importance whether the reservation contains services that are already active or whether all services are still waiting to be started.

To cancel a reservation, the NSP looks up the intradomain reservations that were made for the input reservation and sends a CancelReservation message with the corresponding ID to each of the domains.

5 Conclusions

This deliverable describes the Network Service Plane (NSP) and its northbound interfaces. Transparently enabling multidomain advance reservation features, the NSP allows a network consisting of multiple administrative domains managed by different NRPSs to be integrated into a (Grid-)Meta-Scheduling System.

To achieve this, the NSP offers a reservation webservice that is comparable to a NRPS webservice. Requests received across this interface are processed in such a way that the multidomain nature of the underlying network is hidden as well as possible. Furthermore, the NSP offers a topology webservice that allows administration of the interdomain topology.

6 References

[Hibernate] Hibernate (Relational Persistence for Java and .NET), http://www.hibernate.org/

[JAXB] Java Architecture for XML Binding (JAXB) reference implementation, https://jaxb.dev.java.net/

[MSS] O.Wäldrich, Ph. Wieder, and W. Ziegler, A Meta-Scheduling Service for Co-allocating Arbitrary Types of Resources. In Proc. of the Second Grid Resource Management Workshop (GRMWS'05) in conjunction with Parallel Processing and Applied Mathematics: 6th International Conference, PPAM 2005, Lecture Notes in Computer Science, Volume 3911, R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski (eds.), pp. 782-791, Springer, Poznan, PL, September 11-14, 2005. ISBN: 3-540-34141-2.

[WSN] S. Graham, D. Hull, B. Murray (editors), "Webservice Base Notification 1.3 (WS-BaseNotification)," OASIS Standard, October 2006

[WSS] A. Nadalin, C. Kaler, P. Hallam-Baker, R. Monzill et al., "Webservices Security: SOAP Message Security 1.0 (WS-Security 2004)," OASIS Standard, vol. 200401, 2004.

7 Acronyms

AAA	Authorization, Authentication and Accounting
ARGON	Allocation and Reservation in Grid-enabled Optic Networks
BoD	Bandwidth on Demand
CP	Control Plane
DB	Data Base
DRAC	Dynamic Resource Allocation Controller
E2E	End-to-End
GLIF	Global Lambda Integrated Facility
GMPLS	Generalized Multi Protocol Label Switching
IDM	InterDomain Manager
ID	Identifier
IP	Internet Protocol
I-NNI	Interior NNI
LSP	Label Switched Path
MSS	Meta-Scheduling System
NNI	Network-Network Interface
NRPS	Network Resource Provisioning System
NSP	NSP
NSAP	Network Service Access Point
OSPF	Open Shortest Path First
QoS	Quality of Service
TNA	Transport Network Address
UCLPv2	User Controlled LightPaths version 2
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
WP	Work Package
WS	Webservice