

11.09.06

Deliverable DJ3.4.1: Technical Specification of the Inter- Domain Manager (IDM) Prototype



Deliverable DJ3.4.1

Contractual Date:	30/06/06
Actual Date:	11/09/06
Contract Number:	511082
Instrument type:	Integrated Infrastructure Initiative (I3)
Activity:	JRA3
Work Item:	WI 04
Nature of Deliverable:	R
Dissemination Level	PU
Lead Partner	PSNC
Document Code	GN2-06-184v4

Authors: Mauro Campanella (GARR), Radoslaw Krzywania (PSNC, editor), Afrodite Sevasti (GRNET)

Abstract

This document describes the design specification for the IDM module prototype of the Bandwidth on Demand system developed by the GEANT2 JRA3 activity. The prototype is a proof of concept of the design, information flow control and reservation processing on both inter-domain and intra-domain levels with an initial focus on the Inter-Domain Manager (IDM). The scope of this work is to implement and test the key modules and functionalities, which need to be studied before building a fully operational system. However the prototype is engineered so as to comprise the basis for further system development. It has been designed as a test machine, to allow easy discovery of system weaknesses and potential problems at the early stages of the system development. In the first part of the deliverable, the building blocks of the prototype as organized in packages and classes are documented in full detail, while in the second part the relationships between classes/objects are documented using sequence diagrams.

Table of Contents

0	Executive Summary	v
1	Overview and Implementation Choices	1
2	Static Models – Class Diagrams	4
2.1	Overview of IDM Packages (net.geant2.jra3.*)	4
2.2	net.geant2.jra3.reservation	7
2.2.1	Reservation	8
2.2.2	Service	10
2.2.3	DAO	11
2.2.4	ThreadPoolExecutor	11
2.2.5	User	12
2.3	net.geant2.jra3.interdomain	13
2.3.1	AccessPoint	14
2.3.2	ServiceListener	14
2.3.3	ExternalReservationProcessor	15
2.3.4	ServiceScheduler	15
2.3.5	ExternalReservationProcessorImpl	16
2.3.6	LocalDomain	16
2.3.7	AccessPointService	16
2.4	net.geant2.jra3.interdomain.resources	17
2.5	net.geant2.jra3.communication	18
2.5.1	UserAccessPoint	18
2.5.2	InterdomainCommunication	19
2.5.3	ReservationObserver	20
2.6	net.geant2.jra3.pathfinder.interdomain	22
2.7	net.geant2.jra3.intradomain	22
2.7.1	LocalResourceManager and LocalResourceManagerImpl	23
2.7.2	net.geant2.jra3.intradomain.resources.constraints	24
2.8	net.geant2.jra3.aai	32
2.9	net.geant2.jra3.pnetwork	32
2.9.1	Path and PathImpl	33

2.9.2	Link and LinkImpl	34
2.9.3	Port, ClientPort, and NRENPort	34
2.9.4	Domain, ClientDomain, and NRENDomain	35
2.10	Logging	35
3	Behavioural model – sequence diagrams	38
3.1	Sequence diagram for user's request submission	39
3.2	Sequence diagram for request status check	42
3.3	Sequence diagram for user's service cancel request	44
3.4	Sequence diagram for service schedule processing	46
3.5	Sequence diagram for reservation processing in domain somewhere along reservation path	49
4	Conclusions	51
5	References	52
6	Acronyms	53

Table of Figures

Figure 1.1:	References between modules and packages	2
Figure 2.1:	Overview class diagram for the JRA3 IDM prototype (Part A)	5
Figure 2.2:	Overview class diagram for the JRA3 IDM prototype (Part B)	6
Figure 2.3:	Class diagram for net.geant2.jra3.reservation java package	7
Figure 2.4:	Class diagram for net.geant2.jra3.interdomain java package	13
Figure 2.5:	Class diagram for net.geant2.jra3.communication java package	18
Figure 2.6:	InterdomainPathfinder class	22
Figure 2.7:	Class diagram for net.geant2.jra3.intradomain java package	22
Figure 2.8:	Class diagram for net.geant2.jra3.intradomain.resources.constraints	24
Figure 2.9:	Constraints hierarchy	25
Figure 2.10:	Class diagram for net.geant2.jra3.aai java package	32
Figure 2.11:	Class diagram for net.geant2.jra3.pnetwork java package	33

Figure 3.1: Sequence diagram for user's request submission (Part A)	39
Figure 3.2: Sequence diagram for user's request submission (Part B)	40
Figure 3.3: Sequence diagram for request status check	42
Figure 3.4: Sequence diagram for user's service cancel request	44
Figure 3.5: Sequence diagram for service schedule processing	46
Figure 3.6: Sequence diagram for reservation processing in home domain (Part A)	47
Figure 3.7: Sequence diagram for reservation processing in home domain (Part B)	48
Figure 3.8: Sequence diagram for reservation processing in domain somewhere along reservation path (Part A)	49
Figure 3.9: Sequence diagram for reservation processing in domain somewhere along reservation path (Part B)	50

0 Executive Summary

This document provides the design specification for the Inter-Domain Manager (IDM) prototype of the Bandwidth on Demand system developed by the GEANT2 JRA3 activity. For a detailed overview of the JRA3 Bandwidth on Demand architecture and system the reader should refer to GN2 project deliverable DJ3.3.1 [GN2DJ331]. For the functional specification of the IDM module prototype and of the Inter-domain Manager, the reader should refer to GN2 project deliverable DJ3.3.2 [GN2DJ332].

The prototype is designed as a proof of concept for the design, data and control flow and reservation processing on both inter-domain and intra-domain levels of the JRA3 BoD system with initial focus on the IDM. The scope is to implement and test the key modules and functionalities, which needs to be studied before building a fully operational system. However the prototype is engineered in such a way to comprise the basis for further system development.

The prototype has been designed to be a test machine, to allow easy discovery of system weaknesses and potential problems at early stages of system development. The implementation language is Java in order to ensure interoperability and possibly code reuse with other services being developed in the GN2 project and also with other software. Care has been devoted to logging all events for debugging and code profiling.

In the first part of the deliverable, the building blocks of the prototype as organized in packages and classes are documented in full detail. Organization of classes in packages is made so that different packages exist for the business logic of reservations, the pathfinding functionality, the AAI functionality, the representation of network objects, the inter-domain and user-to-system communication and the intra-domain management, included in the IDM prototype to model the transactions with individual domain managers. In the second part of the deliverable the relationships between classes/objects are documented using sequence diagrams involving objects and methods to model the processes of accepting a user request, checking the status of a reservation, canceling a reservation, service schedule processing and reservation processing both in the home domain and in the rest of domains along the inter-domain path.

The deliverable provides fully detailed documentation of the IDM prototype including all the information required by the reader in order to understand its implementation specifics. It indicates the parts of the prototype implementation that have been introduced with minimal functionality and serve as placeholders for the next phases of implementation, as well as the parts of the current functionality of the prototype that will be moved to the JRA3 BoD system Domain Manager module (see section 5.4.2 of [GN2DJ3.3.1]), when this is implemented.

1 Overview and Implementation Choices

The IDM prototype was coded using the Java language, version 1.5. The objective is to implement the minimum required functionality for a working system, in order to find any possible processing bottlenecks or potential failure points. The prototype fulfils the following functionalities:

- accepting user service requests
- creating service/reservation instances
- executing service and reservation processing (delegated from both – end user and BoD system)
- performing communication tasks to simulate reservation process (no real reservation is done)
- responding to pathfinding queries with pre-defined paths and constraints
- AAI responds always with “OK” message

In order to improve implementation process of the IDM prototype, the following implementation decisions were made:

- all data are stored in volatile memory (no DB engine is used by the prototype IDM version)
- IDM communication is realized with Web Service interfaces
- all other modules are directly hard coded as standalone applications

Regarding software design practices, high level functional blocks are decomposed into smaller entities called Java packages. Those packages help to organize source code, and bring order to classes, interfaces and their dependencies. Each package provides some sub-functionality or structures required by functional blocks to work. Mapping of functional blocks into packages is shown in the **Figure 1.1** below.

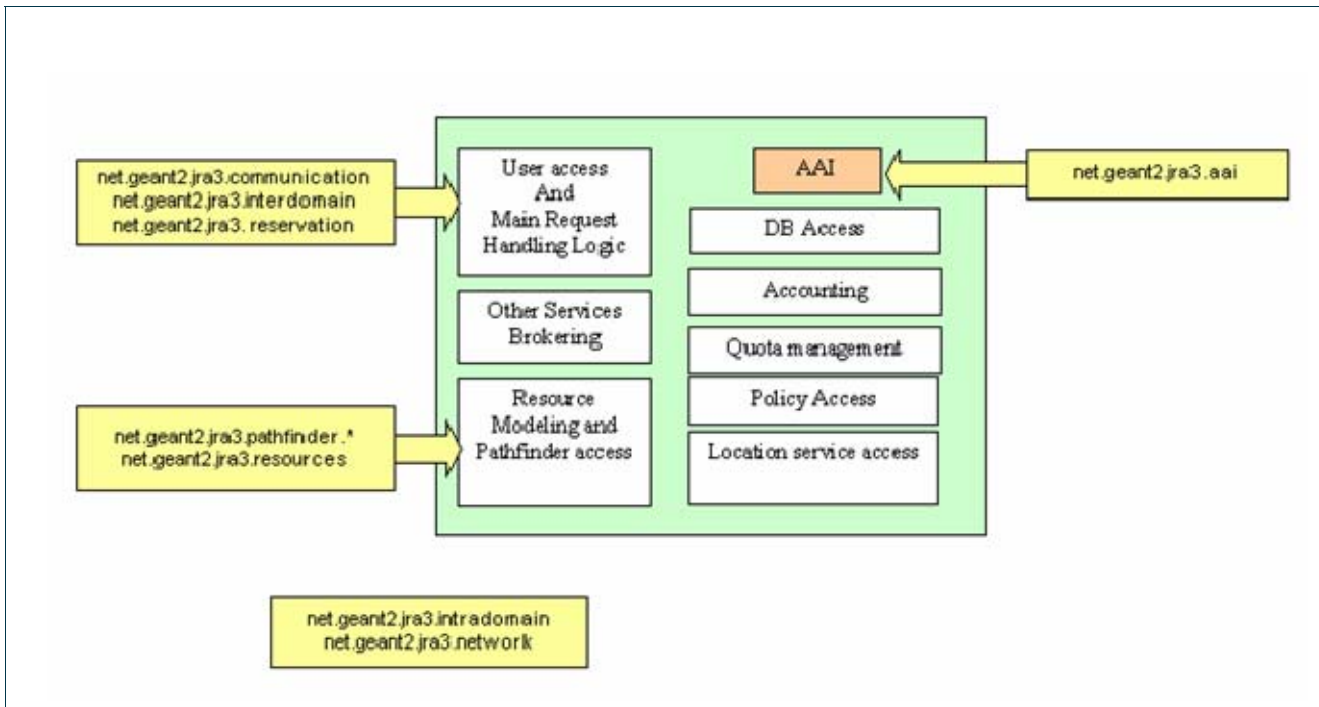


Figure 1.1: References between modules and packages

Packages provide the following functionalities:

- User access and Main Request Handling Logic
 - *net.geant2.jra3.communication* – inter-domain communication using Web Services (IDM-to-IDM protocol) and user access point (limited GUI interface)
 - *net.geant2.jra3.interdomain* – general services and reservations handling
 - *net.geant2.jra3.reservation* – services and reservations' structure and logic (including transaction management)
- Resource Modelling and Pathfinder access
 - *net.geant2.jra3.pathfinder.** - access to pathfinder boxes
 - *net.geant2.jra3.resources* – reserved resource tracking on abstract inter-domain level
- AAI
 - *net.geant2.jra3.aai* – access to AAI box
- non-IDM modules

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

- *net.geant2.jra3.intradomain* – local domain reservations– (intra-domain functionality)
- *net.geant2.jra3.network* – utility package to represent internal network views and topologies

2 Static Models – Class Diagrams

2.1 Overview of IDM Packages (net.geant2.jra3.*)

The diagram depicted in **Figure 2.1** presents an overview of the IDM prototype packages and their relationships. The business logic of reservations is enclosed in packages *net.geant2.jra3.reservation* and *net.geant2.jra3.interdomain*. The more detailed description of the packages is presented in following chapters. Pathfinder package (*net.geant2.jra3.pathfinder.interdomain*) realizes pathfinder functionality, based on the abstract inter-domain network representation. A routing process is not implemented in the prototype release, as it requires more analysis and decisions.

The package *net.geant2.jra3.network* contains classes to represent network objects. The class and its relationships here are based on the abstract network representation design as presented in [GN2DJ332].

The package *net.geant2.jra3.communication* provides a set of interfaces for inter-domain communication and user-to-system communication.

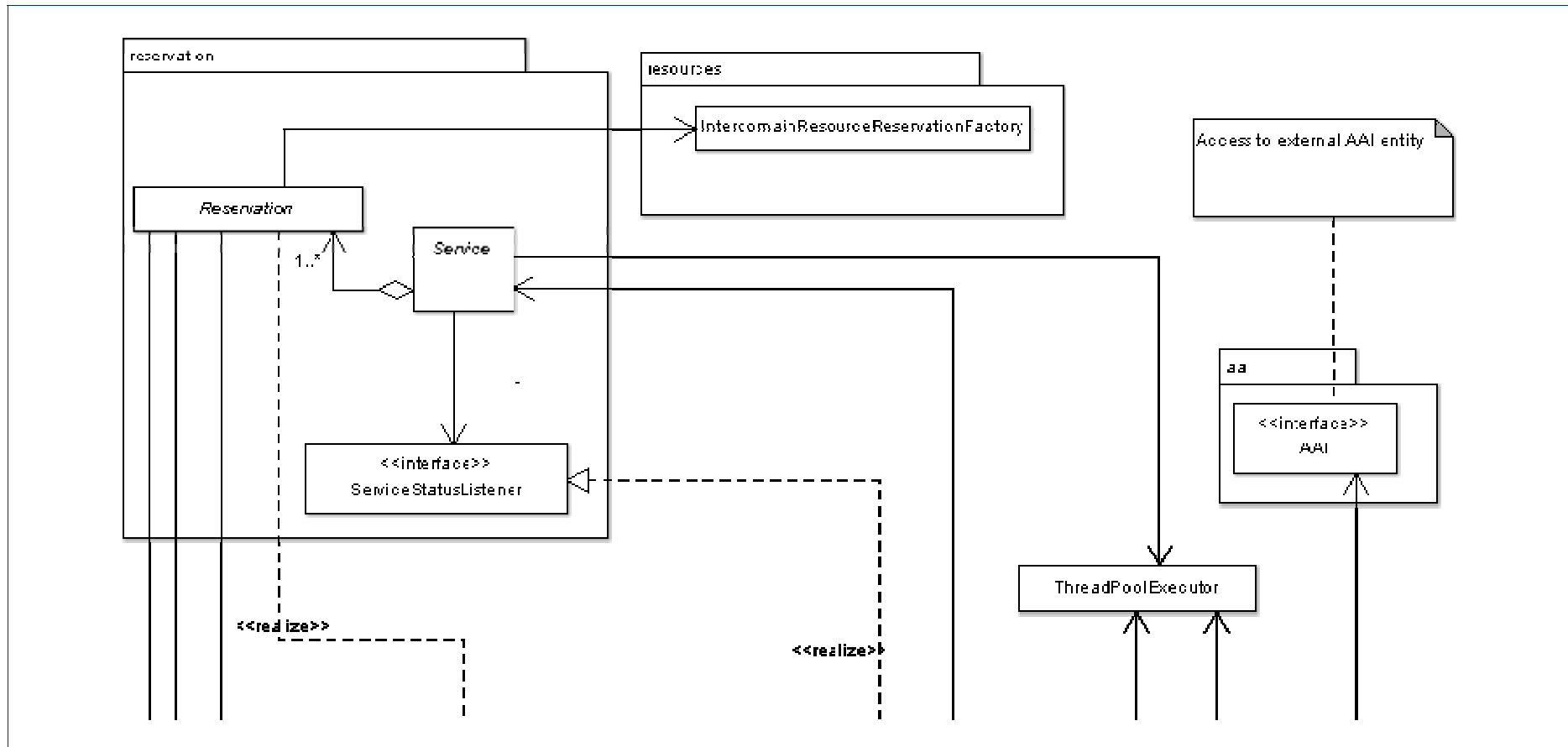


Figure 2.1: Overview class diagram for the JRA3 IDM prototype (Part A)

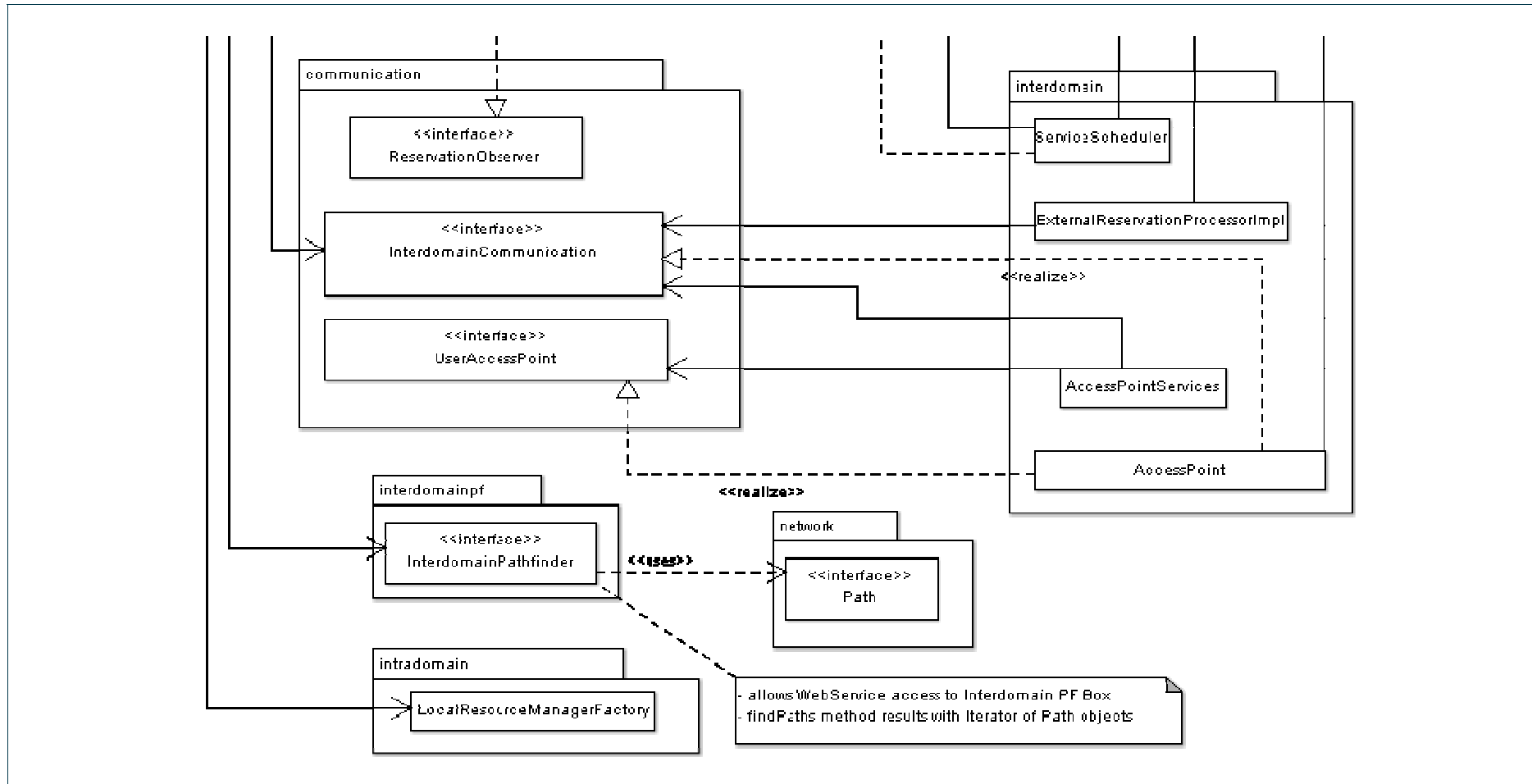


Figure 2.2: Overview class diagram for the JRA3 IDM prototype (Part B)

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

2.2 net.geant2.jra3.reservation

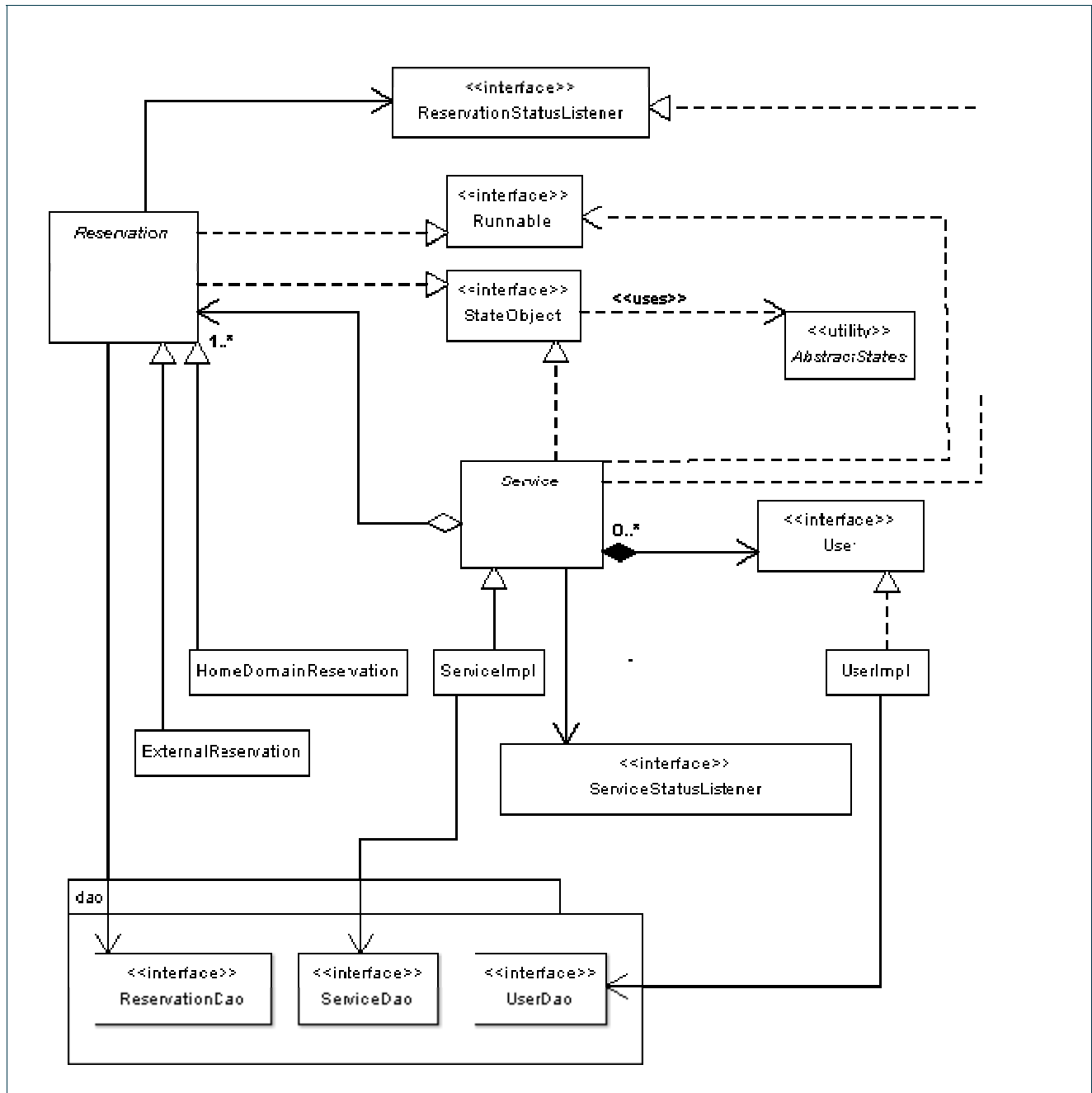


Figure 2.3: Class diagram for `net.geant2.jra3.reservation` java package

The `reservation` package is responsible for the logic of the service and reservations' processing. The base classes of this package are `Reservation` and `Service`. Both of them inherit from the `StateObject` class, which provides information about reservation and service states (state definitions are enclosed in `AbstractStates` class).

2.2.1 Reservation

The `Reservation` class contains fields and methods required to represent and process a user request in the BoD system:

- `getReservationId():String` – returns globally unique identifier of this reservation.
- `getStartPortId():String` – returns identifier of network port where the reservation should start.
- `getEndPortId():String` – returns identifier of network port where the reservation should end.
- `getCapacity():long` – returns requested capacity to reserve.
- `getStartTime():Calendar` – returns time and date when the reservation should start.
- `getEndTime():Calendar` – returns time and date when the reservation should end.
- `getUserData():User` – returns user information who submitted the request.
- `setPath(path:net.geant2.jra3.pnetwork.Path)` – sets reservation path that will be considered for this reservation in negotiation phase.
- `getPath():net.geant2.jra3.pnetwork.Path` – returns reservation path submitted by `setPath()` method.
- `getHomeDomain():String` – returns home domain identifier, where home domain is the domain where reservation was created (in fact it is also the user home domain for the IDM prototype).
- `setResourceReserved(domain:String, reserved:Boolean, message:String):void` – confirms resource reservation in corresponding domains along the reservation path. Parameters:
 - `domain` is identifier of the domain which confirms reservation
 - `reserved` indicates whether resources are reserved
 - `message` contains description of failure in case of problems with resource reservation.

- `cancel()` – cancels reservation on user request.
- `getGlobalConstraints():GlobalConstraints` – returns global constraints for reservation, that are collected and agreed by all domains along the reservation path.
- `setGlobalConstraints(constraints:GlobalConstraints):void` – sets global constraints for reservation, that were agreed by all domains along the reservation path. Parameters:
 - `constraints` is the global constraints variable to be set.
- `getHomeDomainTime():Calendar` – returns current time in home domain, which is the time in the home domain time zone.

Abstract `Reservation` is implemented by two classes:

- home domain reservation (`HomeDomainReservation` class) – this class represents reservation at home domain BoD system and contains full logic for execution of all processes of reservation, including negotiations and decision taking; Objects of this class are stateful and can be serialized to database for failure recovery (notice that the prototype uses volatile memory instead of an external DB engine);
- external reservation (`ExternalReservation` class) – this class represents reservation at all domains along a reservation path, except for the home domain; it contains limited logic, required to retrieve path constraints from local resource manager, and schedule a local reservation, requested by other BoD systems; Objects of this class are stateful and can be serialized to the database for failure recovery (notice that the IDM prototype uses volatile memory instead of an external DB engine);

Reservation may be in following states:

- `accepted` – reservation request is accepted by the system after AAI verification, and awaits execution
- `in progress` – reservation request is under analysis, which involves pathfinding, inter-domain negotiations, resource reservation
- `scheduled` – reservation is scheduled when all resources along the reservation path are booked for the time that the reservation will be used
- `active` – reservation is configured on network devices, and the user is able to use all requested resources
- `finished` – reservation is finished once reservation end time is reached and network devices release allocated resources
- `cancelled` – reservation is cancelled by user request

- failed – a reservation fails if any activity associated with the reservation is finished with error, there are not enough resources, or if the reservation can not be performed for other reasons

The `ReservationStatusListener` interface is associated with a reservation. The interface has to be implemented by classes, which wants to be notified about reservation status changes. The `Reservation` object implements this listener, as it is required to execute reservations sequentially.

2.2.2 Service

`Service` class contains fields and methods required to represent and process user requests in the BoD system:

- `getServiceId():String` – returns a globally unique service identifier.
- `getJustification():String` – returns a justification in human readable form.
- `getPriority():int` – returns priority of the service as requested by the user.
- `addReservation(reservation:Reservation)` – adds new reservation to the service.
Parameters:
 - `reservation` – new `Reservation` object to add.
- `getReservations():Iterator` – returns `java.util.Iterator` object with all the reservations defined for this service.
- `getUserData():User` – returns information on the user who submitted the service.
- `cancel()` – cancels the request and all the corresponding reservations upon user demand.

The `Service` is implemented by the single `ServiceImpl` classes and is used only at the home domain BoD system, where the service was submitted. Domains along a reservation path do not know anything about the service itself, and are focused on single reservations. Therefore the `Service` object at the home domain is responsible for managing the reservation group. If any reservation within the service fails, all already scheduled reservations should be aborted and no more reservations within the request should be executed. A service may be in the following states:

- accepted – service request is accepted by the system after the AAI verification, and awaits execution
- in progress – service request is under analysis, which involves pathfinding, inter-domain negotiations, and resource reservation for each reservation
- scheduled – service is scheduled when all its reservations are scheduled

- active – service is active when all its reservations are active
- finished – service is finished once all its reservations are finished
- canceled – service is cancelled by the user request
- failed – service fails if any reservation associated with service fails (then all reservations should be aborted).

The `ServiceStatusListener` interface is associated with a service and has to be implemented by classes which are interested in notification about service processing completion. `ServiceScheduler` implements this interface, as it is required to execute services sequentially.

2.2.3 DAO

All main classes (`Reservation`, `Service` and `User`) have their own DAO (Data Access Object - `ReservationDao`, `ServiceDao` and `UserDao`) class which is responsible for the creation and the serialization of the objects to the system's database.

Each DAO object consists of the following methods:

- `insert()` – serializes a new object into the database
- `update()` – updates an already existing object in the database .
- `create()` – creates a new object, which is not filled with all required data and is not currently serialized into database.
- `get()` – returns an object of a specified type.

DAO objects implementation should provide guarantees that objects returned by `get()` method are the same in sense of memory presence (the equality-based `Object.equals()` method is insufficient, so singleton design pattern or direct comparison with '=' operator should be used). This will assure that operations in a multithreaded environment are performed on the same objects and database serialization is done correctly.

2.2.4 ThreadPoolExecutor

Service and reservations operate as separate threads. In order to simplify pool of threads management, the `java.util.concurrent.ThreadPoolExecutor` class is used. Both `Service` and `Reservation` classes implement `java.lang.Runnable` interface, that allows execution in separate threads, and especially this can be done under the control of the thread pool manager with `ThreadPoolExecutor.execute(command: java.lang.Runnable)` method. This thread will execute only

the code that is in `java.lang.Runnable:run()` method, all service and reservation logic is written there, including transaction issues. If a method's code is growing and makes the class difficult to read, a decomposition to smaller private methods should be considered. The sequence of method invocation is depicted on sequence diagram in section 3.

2.2.5 User

The `User` class represents a user for the BoD system. For prototype purposes user data are limited to minimum, as they are closely related to AAI (which also is limited) and accounting/quota module (which is not part of the prototype). It consists of two methods:

- `getName():String` – returns user identifier
- `getHomeDomain():String` – returns identifier of the user's home domain.

2.3 net.geant2.jra3.interdomain

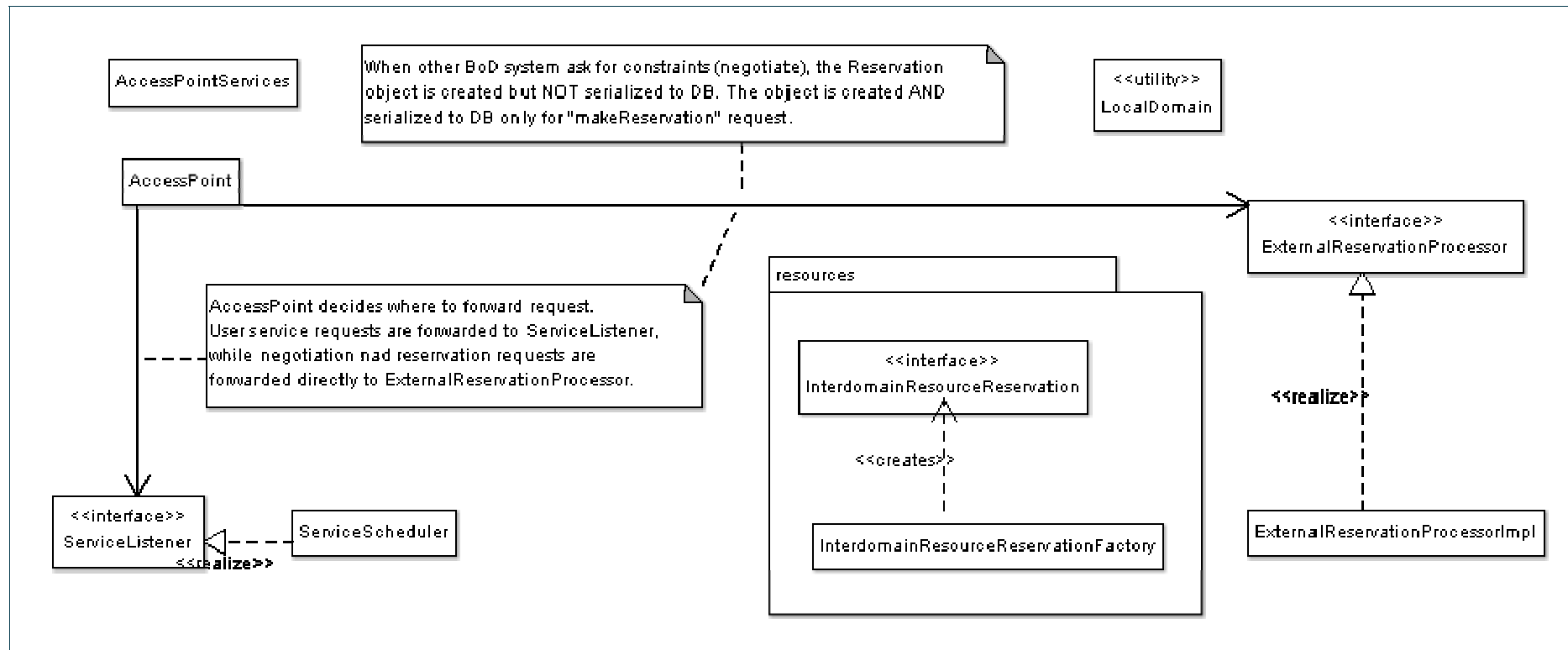


Figure 2.4: Class diagram for net.geant2.jra3.interdomain java package

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

The classes included in `interdomain` package initialize the service or reservation processing. They are responsible for communication both, between the domains and with the users.

2.3.1 AccessPoint

The `AccessPoint` class implements two communication interfaces `net.geant2.jra3.communication.UserAccessPoint` and `net.geant2.jra3.communication.Inter-domainCommunication`, and thus provides send/receive functionality for the IDM-to-IDM communication (see overview of `net.geant2.jra3` UML class diagram). Whatever incoming communication is received (through any of two mentioned interfaces) authentication and authorization are required with the assistance of the AAI interface (see `net.geant2.jra3.aai` package for more details). Only positively verified messages are forwarded to modules/objects responsible for further processing. In addition to the methods from interfaces of `net.geant2.jra3.communication` package, the `AccessPoint` contains the `addServiceListener(sl:ServiceListener)` method. This method adds to the `AccessPoint` the listener whose task is to receive messages from users related to service processing. The `AccessPoint` is responsible for proper delegation of event/messages to corresponding objects/modules. Service requests from a user should be delegated to the `ServiceListener` implementations, while reservation requests received from neighbor BoD systems should be delegated to the `ExternalReservationProcessor` implementations.

The `AccessPoint` provides its functionality with some satellite classes responsible for e.g. web services processing, which are not included in the UML diagrams (as they depend on used technology, schemas, and moreover most of them are generated automatically by a web services' framework like Axis).

2.3.2 ServiceListener

The Interface `ServiceListener` should be implemented by objects which are interested in receiving user requests regarding services. It contains the following methods:

- `cancelService(serviceId:String)` – request to cancel the service and all the corresponding reservations. Parameters:
 - `serviceId` – globally unique service identifier.
- `getServiceStatus(serviceId:String):int` – requests the current service status and immediately returns status value. Parameters:
 - `serviceId` – globally unique service identifier.
- `acceptService(service:String)` – request to execute the service. Parameters:
 - `serviceId` – globally unique service identifier.

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

This interface is implemented by the `ServiceScheduler` class.

2.3.3 ExternalReservationProcessor

The Interface `ExternalReservationProcessor` realizes similar functionality as the `ServiceListener`, but in the context of reservations events received from a neighbor domain's BoD system, rather than user services. It contains the following methods:

- `getConstraints(reservation:net.geant2.jra3.reservation.Reservation, path:net.geant2.jra3.pnetwork.Path)` – this method is invoked once a neighbor BoD system wants to ask for the local domain constraints regarding a specific reservation. In prototype this method will not be used, as BoD systems are asking for a reservation immediately rather than asking for constraints as a first step. Parameters:
 - `reservation` – object with all attributes of a reservation
 - `path` – reservation path.
- `makeReservation(reservation:net.geant2.jra3.reservation.Reservation, path:net.geant2.jra3.pnetwork.Path, constraints:GlobalConstraints)` – this method is invoked when a neighbor BoD system requests for reservation. Parameters:
 - `reservation` – object with all attributes of reservation.
 - `path` – reservation path.
 - `constraints` – currently collected constraints from previous domain along inter-domain path. This domain should add its own local domain constraints to this set before forwarding the request.
- `cancelReservation(res:String):void` – this method is invoked once a user requests for a service cancellation (each reservation then is cancelled one by one). Parameters:
 - `res` – identifier of reservation to cancel.

This interface is implemented by `ExternalReservationProcessorImpl` class.

2.3.4 ServiceScheduler

The `ServiceScheduler` class is responsible for management of multiple instances of a service within a single domain. The primary objective of that is to queue all newly arrived services, and execute them sequentially, so

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

than no more than one service is analyzed at the same time. A service can be executed once the previous one was finished due to successful scheduling or failure. The services' queue should allow to prioritize requests regarding priorities assigned by user (or potentially modified by system at the moment of service acceptance). Services are executed under control of the `java.util.concurrent.ThreadPoolExecutor` with the `execute()` method. Each `Service` class implements the `Runnable` interface, and once method `run()` is executed, the object will be running in a separate thread. Once a service is scheduled or fails, it will notify the registered `ServiceStatusListener` which is implemented by `ServiceScheduler`.

2.3.5 ExternalReservationProcessorImpl

This class is the implementation of `ExternalReservationProcessor`. Its responsibility is focused on reservation requests received from a neighbor domain BoD system. As the `ServiceScheduler` uses the `Service` and indirectly the `HomeDomainReservation` objects, this class uses the `ExternalReservation` objects instead. The `ExternalReservationProcessor` has no knowledge about the service, to which a reservation belongs, as a service responsibility remains only at a home domain of a request. A reservation controlled by this class should be executed simultaneously.

2.3.6 LocalDomain

The `LocalDomain` class is an informational entity that provides information about the local domain. It should be implemented with following functionality:

- `getLocalDomainId():String` – returns the globally unique identifier of this domain. This method needs to be implemented as static.
- `getConstraintsCMLPath():String` – returns a path to a file on a local hard drive, which contains information about domain constraints. This method is used for prototype purpose only, and will be removed in further phases of development.
- `getPathsXMLPath():String` – returns a path to a file on a local hard drive, which contains information about inter-domain paths. This method is used for prototype purpose only, and will be removed in further phases of development.
- `getTopologyXMLPath():String` – returns a path to a file on a local hard drive, which contains information about inter-domain topology. This method is used for prototype purpose only, and will be removed in further phases of development.

2.3.7 AccessPointService

This class contains static reference points to classes, which are used by multiple entities. The mechanism used here allows a single point of access to such classes, which simplifies the source code and improves its readability.

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

2.4 net.greant2.jra3.interdomain.resources

The subpackage `resources` contains the interface `InterdomainResourcesReservation` the objective of which is to store currently booked inter-domain resources. It is used to check the availability time-variant resources for inter-domain interconnections. In the prototype only the capacity is taken into consideration. Implementation of this interface should provide the following functionality:

- `addReservation(linked:String, reservation:Reservation)` – adds new reservation to the booked resources. Parameters:
 - `linked` – identifier of network link to which this reservation is assigned and will consume resources.
 - `reservation` – object with reservation attributes, including capacity and start/end time.
- `removeReservation(reservation:Reservation)` – removes reservation from resources. Parameter:
 - `reservation` – object with reservation attributes.
- `checkMaxUsage(linked:String, startTime:Calendar, endTime:Calendar):long` – this method returns highest value of reserved capacity over a specified link between specified date/time. Parameters:
 - `linkId` – identifier of interdomain link for which maximum usage should be calculated.
 - `startTime/endTime` – start and end time of the period for which calculations should be performed.

The `InterdomainResourcesReservation` is not dedicated to intra-domain resource usage tracking. Its purpose is to keep limited knowledge only about inter-domain links.

In order to initialize the `InterdomainResourcesReservation` object, the `getInstance()` method of the `InterdomainResourceReservationFactory` class should be used.

2.5 net.geant2.jra3.communication

This package provides interfaces for communication, including both User-to-IDM and IDM-to-IDM protocols.

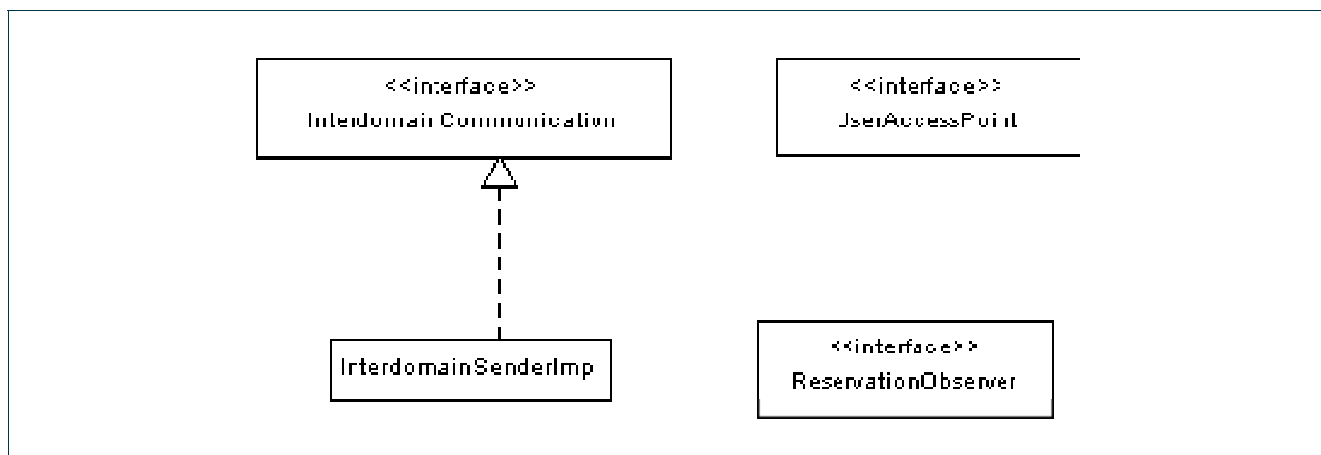


Figure 2.5: Class diagram for net.geant2.jra3.communication java package

2.5.1 UserAccessPoint

The `UserAccessPoint` is the class which accepts user requests. In the IDM prototype, requests are loaded from static XML files. There is a need to provide a simple text or graphical interface for this class in order to allow choosing between various file names, with service, check status and cancellation requests. The `UserAccessPoint` supports the following methods:

- `acceptService(serviceDesc:String):String` – should perform the following operations in order of presence : convert XML service description using proper parser, extract user information, create `User` object, authenticate and authorize user in AAI, and eventually create `Service` and related `Reservation` objects regarding the request. The created objects receive globally unique identifiers and are forwarded to `ServiceScheduler` through `ServiceListener` interface for service execution. As a result of this method the service identifier is returned to the user. Parameters are:
 - `serviceDesc` – String with XML representation of service request
- `cancelService(serviceDesc:String)` – after authentication and authorization of the user for this request, a proper `Service` object should be obtained via `ServiceDao` using the identifier provided. Once a `Service` object is available, a `Service.cancel()` method should be invoked. As a service cancellation may take long time and involve multi-domain cooperation, this method returns to its caller method immediately, and the user is notified via e-mail about the service finish. Parameters:

- `serviceDesc` – globally unique service identifier.
- `checkServiceState(serviceId:String) : int` – after the authentication and authorization of the user for this request, a proper `Service` object should be gotten via `ServiceDao` using the identifier provided. Once a `Service` object is available, a `Service.getState()` method should be invoked and the result value is forwarded back to user. Parameters:
 - `serviceDesc` – globally unique service identifier.

2.5.2 InterdomainCommunication

The *InterdomainCommunication* is an interface to send/receive information to/from other BoD systems. To send the information which requires a response, a proper observer must be specified, in order to be notified about the response message. Observers should be implemented by classes inheriting the *net.geant2.jra3.reservation.Reservation* class, so the responses always can be addressed to the message sender. The *InterdomainCommunication* class implements the following methods:

- `getConstraints(reservation:net.geant2.jra3.reservation.Reservation, constraints:GlobalConstraints)` – this method is used to send constraints request to the next domain on the inter-domain path. This method is not used in the prototype as domains request reservations immediately, rather than asking beforehand for potential constraints. Parameters:
 - `reservation` – Reservation object with all requested attributes of reservation.
 - `constraints` – global constraints that must be agreed along all domains on the reservation path.
- `reportConstraints(reservation:net.geant2.jra3.reservationReservation, constraints:net.geant2.jra3.intradomain.resources.constraints.GlobalConstraints)` – this method is used to send a constraints' report to the previous domain on the inter-domain path. This method is not used in the prototype as domains request reservations immediately, rather than asking beforehand for potential constraints. Parameters:
 - `reservation` – a reference to reservation which will be notified about a received report.
 - `constraints` – local domain constraints that should be reported.
- `scheduleReservation(reservation:net.geant2.jra3.reservation.Reservation, observer:ReservationObserver)` – this method is used to send the reservation schedule requests to the next domain on inter-domain path. Parameters:
 - `reservation` – Reservation object with all requested attributes for reservation.
 - `observer` – object that should be notified when response to this request arrives.

- `reportSchedule(reservation:net.geant2.jra3.reservation.Reservation, message:String, success:boolean)` – this method is used to send a reservation schedule response with confirmation or a failure report to the previous domain on the inter-domain path. Parameters:
 - `reservation` – Reservation object which previously send a reservation schedule request, and should be notified about this response.
 - `message` – contains human readable description of the schedule report, which is obligatory to use with the failure reports
 - `success` – indicates whether the reservation schedule was a success or failure.
- `cancelReservation(reservation:net.geant2.jra3.reservation.Reservation, reservationObserver:ReservationObserver)` – sends request for reservation cancellation to the neighbor BoD domain. Parameters are:
 - `reservation` – Reservation object with all the information about the reservation to be cancelled
 - `reservationObserver` – the observer that will be waiting for the response to this request.

2.5.3 ReservationObserver

This interface should be implemented by objects, which are interested in receiving response messages for inter-domain requests. It includes the following functionality:

- `reservationScheduled(success:Boolean, message:String)` – reports reservation schedule from neighbor BoD domain, as a response for a previously sent request. Due to the chained communication model, the report also indicates that reservation is scheduled in all the domains along the reservation Path, that are preceding the current domain. Parameters are:
 - `success` – indicates whether the reservation is successfully scheduled
 - `message` – provides human readable description, used mostly for providing the failure reasons.
- `reservationCanceled(success:boolean, message:String)` – reports reservation cancellation from neighbor BoD domain, as a response for a previously sent cancel request. Due to the chained communication model, the report also indicates that the reservation is cancelled in all domains along the reservation Path, that are preceding the current domain.
 - `success` – indicates whether the reservation is successfully canceled
 - `message` – provides human readable description, used mostly for providing failure reasons.

- `setGlobalConstraints(constraints:net.geant2.jra3.intradomain.resources.constraints.GlobalConstraints)` – reports `GlobalConstraints` from a neighbor BoD domain, when the reservation results are sent from the last domain to the first domain. As a chained communication model is used, this implies that this method overwrites the current `GlobalConstraints` stored in reservation, with the `GlobalConstraints` that were solved at the last reservation domain and these values should be used for resources' booking. Parameters:
 - `constraints` – final `GlobalConstraints` values that should be used for resource booking

2.6 net.geant2.jra3.pathfinder.interdomain

Pathfinder package includes the `interdomain` sub-package, and provides interface to pathfinder access.

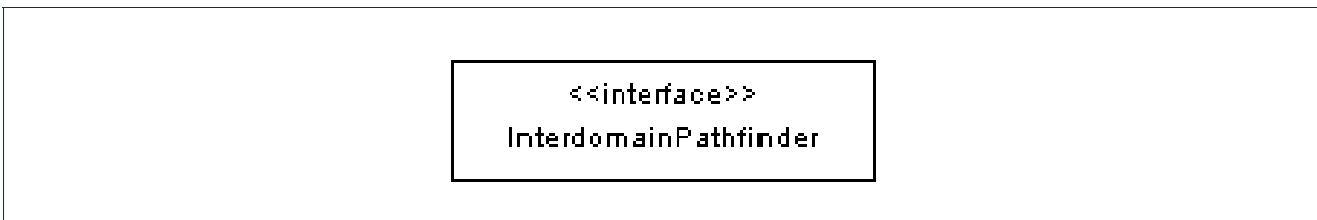


Figure 2.6: InterdomainPathfinder class

The method `findInterdomainPath(reservation:Reservation)` results with an ordered list of possible inter-domain reservation paths, corresponding to the reservation attributes. The paths are of the type `net.geant2.jra3.pnetwork.Path`. Implementation of this interface should provide static return values for path requests, regarding reservation parameters. The set of the paths should be saved as XML files and translated into objects via an XML parser. Usage of XML file format allows scalability for test purposes and will increase flexibility with constructing new test cases. Parameters:

- `reservation` – object with all reservation parameters, that are required during the path finding process.

2.7 net.geant2.jra3.intradomain

This package is responsible for local domain management. For prototype purposes the intra-domain functionality is limited, as the primary objective is to examine the inter-domain BoD behavior.

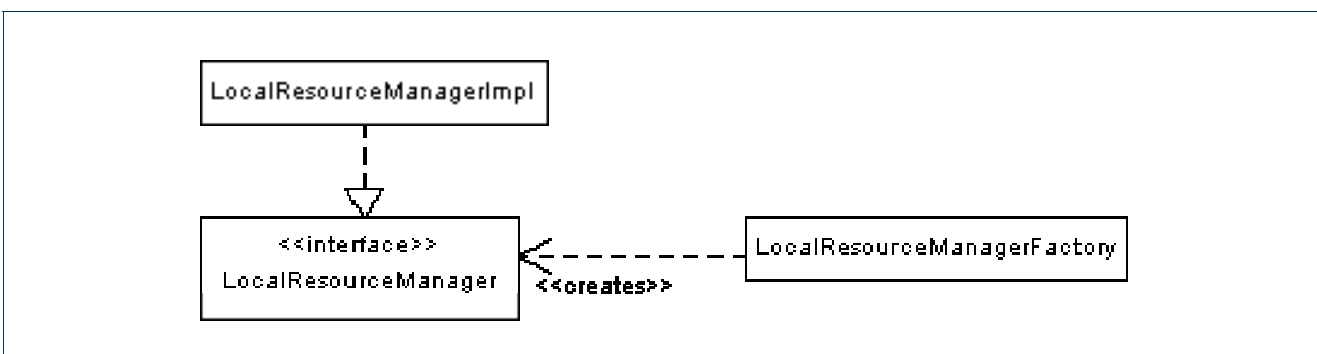


Figure 2.7: Class diagram for net.geant2.jra3.intradomain java package

2.7.1 LocalResourceManager and LocalResourceManagerImpl

The *LocalResourceManager* interface, and its implementation provides methods for checking resource availability in a local domain. A Domain prepares a set of constraints (technology neutral) which should be agreed along all the domains on the inter-domain paths. Those constraints are static values, downloaded from XML files, associated with prepared test cases (test cases includes paths, constraints and reservation examples). The *LocalResourceManager* realizes the following functions:

- `checkResources(path:net.geant2.jra3.pnetwork.Path, constraints:net.geant2.jra3.intradomain.resources.GlobalConstraints)` – checks for the local domain paths that can be configured based on the inter-domain path provided and defines constraints for the potential paths. Constraints need to be added to the `constraints` parameter. Parameters:
 - `path` – inter-domain path for the reservation.
 - `constraints` – already collected local constraints from BoD domains preceding current one.
- `reserveResources(path:net.geant2.jra3.pnetwork.Path, constraints:net.geant2.jra3.intradomain.resources.GlobalConstraints)` – request for local domain resources that correspond to the inter-domain path provided and the agreed constraints. The decision of which local path is chosen (if more than one possible) is left to *LocalResourceManagerImpl* as long as constraints are respected. Parameters:
 - `path` – inter-domain path for reservation.
 - `constraints` – agreed local domain constraints from all BoD domains.

In order to create an instance of the *LocalResourceManager* class, the static *LocalResourceManagerFactory* class should be used, invoking the `getInstance()` method. By default the instance of the *LocalResourceManagerImpl* class is returned.

2.7.2 net.geant2.jra3.intradomain.resources.constraints

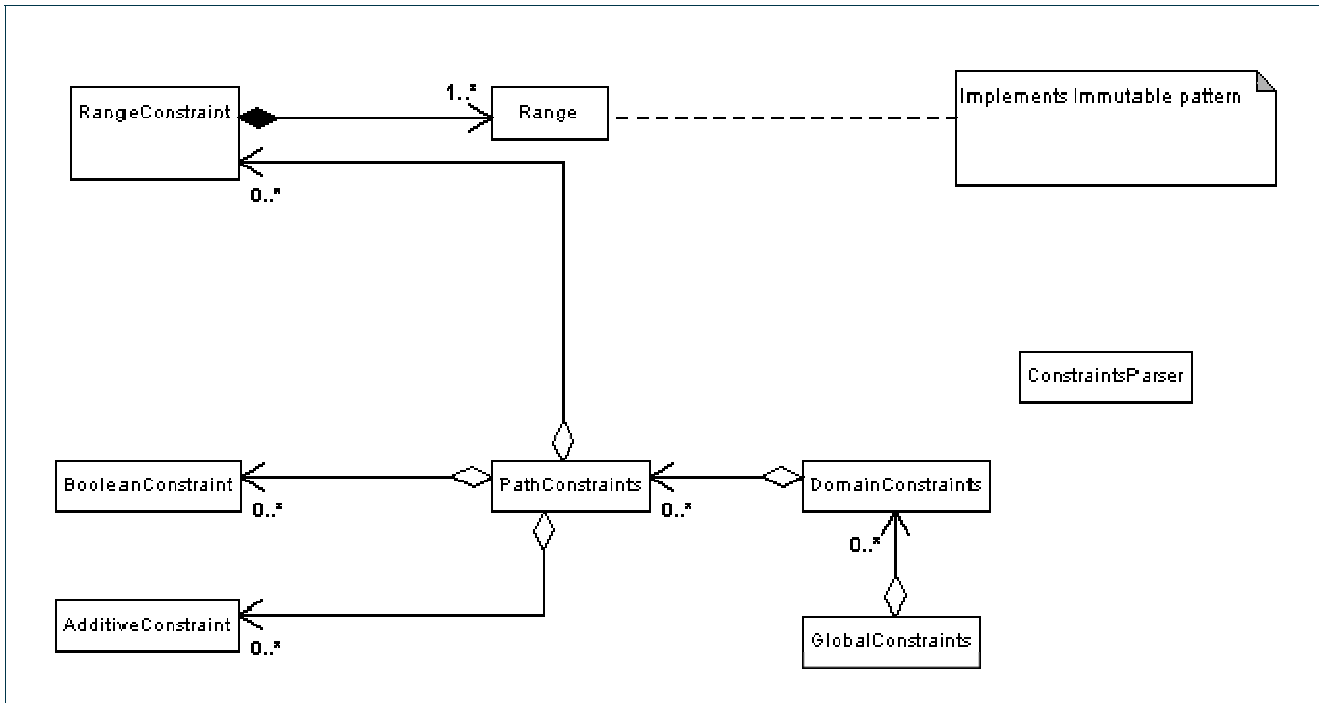


Figure 2.8: Class diagram for net.geant2.jra3.intradomain.resources.constraints

Classes from this package (**Figure 2.8**) represent local domain constraints (mostly technology agnostic) which should be agreed along all domains for reservation. Classes are organized here in the following hierarchy, shown in the **Figure 2.9**. There is one `GlobalConstraints` per reservation, which may consist of multiple `DomainConstraints` – one for each domain on the reservation path. `DomainConstraints` consists of `PathConstraints` – one for each potential intra-domain path. `PathConstraints` may include any of the following basic constraints (each by multiple times) – `AdditiveConstraint`, `BooleanConstraint`, and `RangeConstraint`. Objects contain logic that allows calculating final reservation parameters, without understanding them (therefore it can be done just on the inter-domain level). There are the following logical relations between the objects on hierarchy tree:

- between each `PathConstraints` within `DomainConstraint` is relation OR (any of `PathConstraints` may be finally used as local path),
- between each `DomainConstraints` in `GlobalConstraints` is relation AND (all of them must agree),
- between each constraint within `PathConstraints` is relation AND (all of them must be respected),
- between any `PathConstraint` that belong to different `DomainConstraints` is relation AND (all of them must agree).

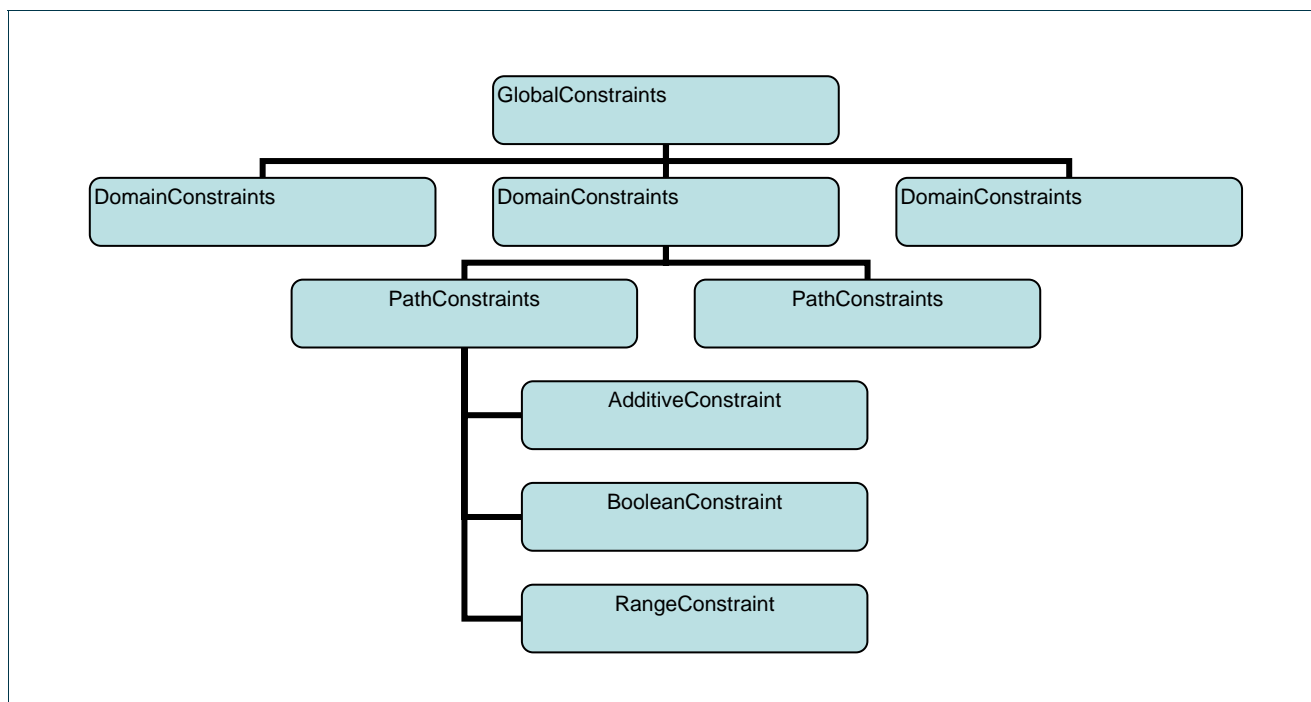


Figure 2.9: Constraints hierarchy

To come up with the final constraint there is a need to use logic implemented in all constraint objects in the hierarchy. As the final product, the single `GlobalConstraints` object is expected with specific final reservation constraints for each domain along the inter-domain path.

Notice: Constraints' classes are placed in the `net.geant2.jra3.intradoman` sub-package, which will be the basis for the Domain Manager (DM) implementation. This approach is valid and sufficient for the prototype only, and requires the attention for the further development phases. As constraints' classes are general, it is possible to model both – technology neutral (like delay, cost, etc.) and technology specific constraints (Ethernet VLAN ranges, capacity, etc.). Therefore, in subsequent implementation phases, the classes will be moved into separate package, independent from intra and inter-domain levels.

2.7.2.1 *GlobalConstraints*

This class represents global constraints for a single reservation. It includes all the domain constraints that need to be agreed for a reservation to succeed. This object is forwarded between the domains in order to collect all the constraints, and also, after determining the actual parameters to be used at the last domain, to make a reservation with the specified parameters. The object contains a logic that enables resolving of the collected constraints into strict parameters for all reservations at a local domain level. The following methods are available:

- `addDomainConstraints(domainId:String, constraints:DomainConstraints)` – adds the new local domain constraints to the global constraints container. Parameters:

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

- `domainId` – identifier of the domain adding the `DomainConstraints`.
- `constraints` – local domain constraints.
- `getDomainConstraints(domainId:String)` – returns constraints for the specified domain. Parameters:
 - `domainId` – identifier of the domain.
- `calculateConstraint():GlobalConstraints` – resolves collected constraints into a set of strict constraints that should be used for the reservations in the individual domains. The resolved object is returned as the result of this method call. For each domain there is one `DomainConstraints` object with such parameters.
- `getDomainConstraints():java.util.List` – returns a list of all domain constraints.
- `setDomainConstraints(domainConstraints:java.util.List)` – adds multiple domain constraints at once. It should be used with `setDomainIds()` method. Parameters:
 - `domainConstraints` – a list of domain constraints to add.
- `getDomainsIds():java.util.List` – returns a list of the identifiers of all the domains, for which constraints are stored in a `GlobalConstraints` object
- `setDomainIds(domainIds>List)` – adds multiple domain constraints at once. It should be used with `setDomainConstraints()` method. Parameters:
 - `domainIds` – a list of the domain identifiers to add.

2.7.2.2 *DomainConstraints*

This class represents local domain constraints for all the potential intra-domain reservation paths. The object contains also logic to find a common part (intersection) of local domain constraints, which is used for a global constraints agreement. The following methods are available:

- `addPathConstraints(constraints:PathConstraints)` – add new constraints for a single potential intra-domain reservation path. Parameters:
 - `constraints` – intra-domain reservation path constraints
- `intersect(constraints:DomainConstraints):DomainConstraints` – limits constraints to the intersection between the current domain's constraints and those of the domain specified. The resulting set of strict reservation parameters is returned. Parameters:

- `constraints` – domain constraints to search for a common part.
- `getPathConstraints():java.util.List` – returns a list of all the included path constraints.
- `setPathConstraints(constraints:java.util.List)` – sets multiple path constraints at once.
Parameters:
 - `constraints` – a list of path constraints to add.
- `copy():DomainConstraints` – clones object.
- `isValid():boolean` – indicates whether this object is valid, which means that there is at least one path constraint that meets the request requirements.

2.7.2.3 PathConstraints

This class represents intra-domain path constraints that should be agreed along all domains at the inter-domain level. At the prototype the number of such parameters is limited, and includes at least:

- delay between ingress and egress ports (as milliseconds)
- capacity (indicates if the requested capacity can be configured)
- resilience (indicates whether the path has some backup options)
- VLAN range (for Ethernet technology only, VLAN number must be agreed by all Ethernet domains along the reservation path)
- Path cost (indicates path cost for further purposes)

`PathConstraints` class contains the following methods:

- `addRangeConstraint(constraint:RangeConstraint, name:String)` – adds a new constraint associated with values range. Parameters:
 - `constraint` – constraint to be added.
 - `name` – name/identifier of the constraint type.
- `addBooleanConstraint(constraint:BooleanConstraint, name:String)` – adds new constraints associated with the true/false values. Parameters:
 - `constraint` – constraint to be added.

- name – name/identifier of constraint type.
- `addAdditiveConstraint(constraint: AdditiveConstraint, name:String)` – adds new constraint associated with the increasing parameter values. Parameters:
 - constraint – constraint to be added.
 - name – name/identifier of constraint type.
- `Intersect(constraints:PathConstraints):PathConstraints` – merges constraints in order to define a common subset of the reservation parameter values. Parameters:
 - constraints – constraints to intersect.
- `getRangeConstraint(name:String)` – returns the range constraints with the specified name. Parameters:
 - name – name of the constraint to retrieve.
- `getBooleanConstraint(name:String)` – returns boolean constraints with the specified name. Parameters:
 - name – name of the constraint to retrieve.
- `getAdditiveConstraint(name:String)` – returns additive constraints with the specified name. Parameters:
 - name – name of the constraint to retrieve.
- `getBooleanConstraintKeys():java.util.List` – returns all key names of boolean constraint type, that are included in the current instance of `PathConstraints` object.
- `getAdditiveConstraintKeys():java.util.List` – returns all key names of additive constraint type, that are included in the current instance of `PathConstraints` object.
- `getRangeConstraintKeys():java.util.List` – returns all key names of range constraint type, that are included in the current instance of `PathConstraints` object.
- `getBooleanConstraints():java.util.List` – returns a list of boolean constraints included in the current instance of `PathConstraints` object.
- `getAdditiveConstraints():java.util.List` – returns a list of additive constraints included in the current instance of `PathConstraints` object.

- `getRangeConstraints() : java.util.List` – returns a list of range constraints included in the current instance of `PathConstraints` object.
- `setAddConstraints(addConstraints : java.util.Map)` – adds multiple additive constraints at once. Parameters:
 - `addConstraints` – the additive constraints to add.
- `setBoolConstraints(boolConstraints : java.util.Map)` – adds multiple boolean constraints at once. Parameters:
 - `boolConstraints` – the boolean constraints to add.
- `setRangeConstraints(rangeConstraints : java.util.Map)` – adds multiple range constraints at once. Parameters:
 - `rangeConstraints` – the range constraints to add.
- `intersectBooleanConstraints(constraints2 : PathConstraints, result : PathConstraints)` – intersects all the boolean constraints by comparing `constraints2` object to current `PathConstraints` instance. The result is written into `result` object. Parameters:
 - `constraints2` – the constraint object to intersect.
 - `result` – the object to write results into.
- `intersectAdditiveConstraints(constraints2 : PathConstraints, result : PathConstraints)` – intersects all the additive constraints by comparing `constraints2` object to the current `PathConstraints` instance. The results is written into `result` object. Parameters:
 - `constraints2` – the constraint object to intersect.
 - `result` – the object to write results into.
- `intersectRangeConstraints(constraints2 : PathConstraints, result : PathConstraints)` – intersects all range constraints by comparing `constraints2` object to current `PathConstraints` instance. The result is written into the `result` object. Parameters:
 - `constraints2` – the constraint object to intersect.
 - `result` – the object to write results into.
- `copy() : PathConstraints` – clones this object.

2.7.2.4 AdditiveConstraint

Stores constraints that have additive nature, which means that values are summed along all domains. This includes constraints such as:

- delay
- cost.

It contains logic to sum with other constraint of the same type. This class provides the following methods:

- `setValue(value:Double)` – sets value of this constraint object to a specified one. Parameters:
 - `value` – value to set.
- `getValue():Double` – returns value of this constraint object.
- `Intersect(additive:AdditiveConstraint):AdditiveConstraint` – calculates sum of the object's own value and value of a specified constraint object. The new value is returned as new `AdditiveConstraint` object. Parameters:
 - `additive` – constraints to sum.

2.7.2.5 BooleanConstraint

Stores constraints that have true/false nature, which means that values are restricted to logical AND/OR operations. This includes constraints such as:

- resiliency

It contains logic to sum with other constraints of the same type. This class provides the following methods:

- `setValue(value:Boolean)` – sets value of this constraint object to a specified one. Parameters:
 - `value` – value to set.
- `getValue():Boolean` – returns value of this constraint object.
- `intersect(boolean:BooleanConstraint):BooleanConstraint` – calculates boolean AND of the object's own value and the value of a specified constraint object. New value is returned as new `BooleanConstraint` object. Parameters:
 - `boolean` – constraint to perform the logical AND with.

2.7.2.6 RangeConstraint

Stores constraints that have a range nature, which means that the values need to be intersected in order to find the common part. This includes constraints such as:

- available VLAN numbers

It contains logic to intersect with other objects of the same type. This class provides the following methods:

- `addRange(min:Integer, max:Integer)` – adds new range to this object. Parameters:
 - `min` – start value of the range,
 - `max` – end value of the range.
- `getRanges():Iterator` – returns all the ranges within this object as Iterator of type `Range`.
- `intersect(range:RangeConstraint):RangeConstraint` – intersect ranges with the objects specified as parameters and return result as a new `RangeConstraint` object. Parameters:
 - `range` – constraint to intersect with.
- `setRanges(ranges:java.util.List)` – sets multiple range constraints at once. Parameters:
 - `ranges` – a range constraints list to be set.
- `removeRange(min:Integer,max:Integer)` – removes range constraints in the range specified by `min` and `max` parameters. Parameters:
 - `min` – starting range value.
 - `max` – ending range value.
- `getCount():int` – returns number of currently stored ranges.
- `isEmpty():boolean` – indicates whether there are currently any ranges defined (value of false is returned), or constraint is empty (value of true is returned).
- `difference(constraint2:RangeConstraint):RangeConstraint` – performs difference operation on current object with `constraint2` object. As a result ranges which exist in `constraints2` objects are removed from the current range constraint. Parameters:
 - `constraint2` – the `RangeConstraint` object for difference operation

2.7.2.7 ConstraintsParser

This class performs XML parsing operations to provide the IDM prototype with the statically defined constraints for the reservations.

2.8 net.geant2.jra3.aai

This package (**Figure 2.10**) represents access to the AAI utility in a single domain. It provides two methods for authenticating (`boolean authenticate(User user)`) and authorizing (`boolean authorize(User user, Action action)`) for a specified action (the action can be service request, status checking, request cancellation or reservation request). In the prototype, the AAI interface is implemented as a mock object (emulator), which always validates a user and his requests positively.

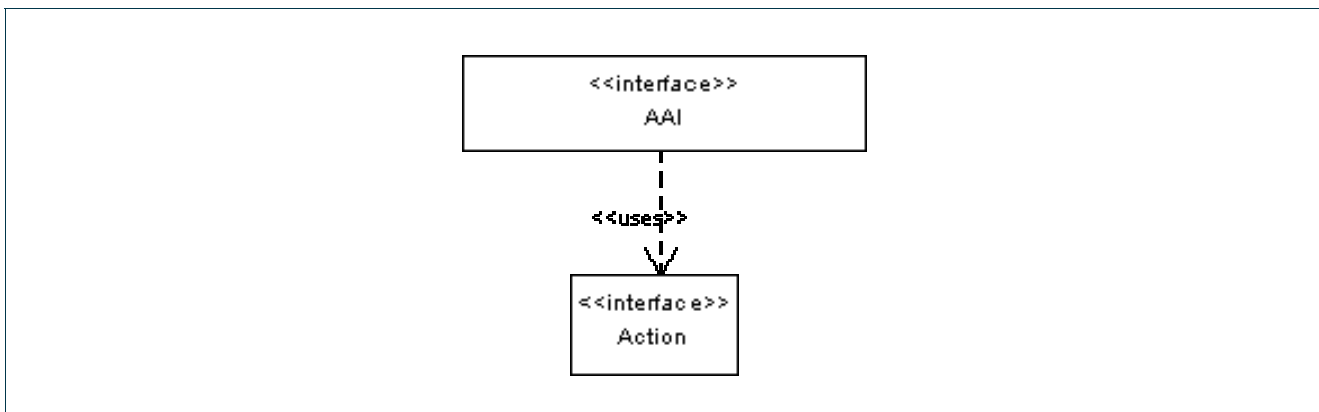


Figure 2.10: Class diagram for net.geant2.jra3.aai java package

2.9 net.geant2.jra3.pnetwork

This package represents the limited network view, and is based on the abstract network representation design as presented in [GN2DJ332]. The classes are limited in comparison to the full schema, as the prototype is focused on the inter-domain level and provides test functionality. The package is named `pnetwork` as for “prototype network”. For future releases there will be the need to redesign it and implement it based on the full abstract representation schema.

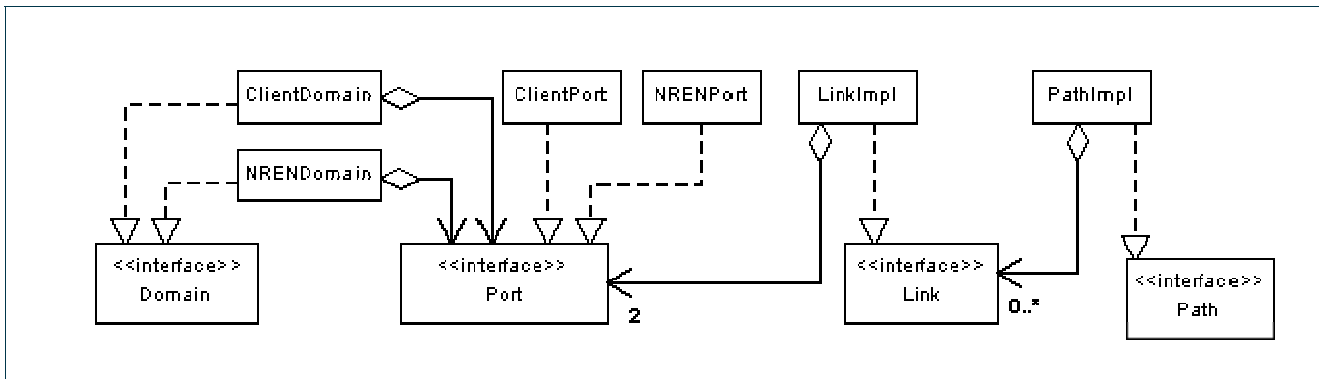


Figure 2.11: Class diagram for net.geant2.jra3.pnetwork java package

2.9.1 Path and PathImpl

Represents an inter-domain reservation path. It provides the following methods:

- `getCapacity() : long` – returns the capacity reserved for this path.
- `getId() : String` – returns the globally unique identifier of this path.
- `getStartingDomain() : String` – returns the globally unique identifier of reservation start domain.
- `getEndingDomain() : String` – returns the globally unique identifier of reservation end domain.
- `getLinks() : Iterator` – returns an ordered iterator of all links that this path includes.
- `getType() : String` – return information about the path type.
- `isResilient() : boolean` – returns true if all links in the path have resiliency options.
- `addLink(link : Link)` – adds new link to this path. Parameters:
 - `link` – new link that will be added to path.
- `removeLink(link : Link)` – removes link from the path. Parameters:
 - `link` – link to be removed from path.
- `addAll(path : Path)` – adds all links from the specified path to the current path. Parameters:
 - `path` – a path to be copied.
- `getIngress(domainId : String) : Link` – returns ingress link to the specified domain. Parameters:

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

- domainId – identifier of a domain to find ingress link into.
- getEgress(domainId:String):Link – returns egress link from the specified domain. Parameters:
 - domainId – identifier of a domain to find egress link from.

PathImpl object has direct references to Link interfaces, of which the path is constructed. This link list can be accessed with getLinks() method.

2.9.2 Link and LinkImpl

Represents an inter-domain link. It provides the following methods:

- getId():String – returns the globally unique identifier of this link.
- getStartPort():String – returns the globally unique identifier of the link's start port.
- getEndPort():String – returns the globally unique identifier of the link's end port.
- getType():String – returns the type of this link.
- getCapacity():long – returns the link's maximum capacity.
- isResilient():boolean – returns true if this link has any resiliency options.

The link has direct references to the pair of involved ports.

2.9.3 Port, ClientPort, and NRENPort

Represents a network port that belongs to inter-domain path. It provides the following methods:

- getId():String – returns the globally unique identifier of this port.
- isClientPort():boolean – indicates whether this port belongs to a start/end client port of a reservation.
- getDomainId():String – returns the globally unique identifier of domain to which this port belongs.
- getTechnology():String – returns the technology which is provided by this port (Ethernet, SDH, etc.).
- getType():String – returns the type of this port (Client or NREN port).

`ClientPort` and `NRENPort` are two implementation of `Port` interface. The first one is related to a client start/end point of reservation. The second indicates that port is a regular port that belongs to NREN domain. Port implementations have direct reference to the `Domain` that they belong to.

2.9.4 Domain, ClientDomain, and NRENDomain

Represents a single domain. It provides the following methods:

- `getId():String` – returns a globally unique identifier of domain.
- `isClientDomain():String` – indicates whether this domain belongs to a client, or is an NREN domain.
- `getPorts():Iterator` – returns the set of all ports that are involved in the inter-domain communication for this domain.

2.10 Logging

The prototype places strong emphasis on logging functionality as the only way to track the system behavior. The Log4J library is used for this purpose. Two types of logging are distinguished:

- error logging
- processing logging

Errors are not very important in the prototype, as long as they do not make the prototype impossible to use. It is assumed that prototype will operate in a perfect environment, where no critical errors, like communication or hardware failures, may occur. Therefore only try/catch sections are equipped with the error logging functionality, as this section may catch some exceptions called by JVM. Logs are written to a file which name starts with “error_” string. XML should not be used as file format for logs, as it is far easier for logs to be read by humans, if they are in the form of data-class-exception-message.

For processing logging also an XML format has also not been used for the same reason. The name of the file for system processing logging must start with “processing_” string. Current time and date should also be included into the file name, if possible, to easily distinguish between multiple log files. The logging is performed on the operations as show in the example in **Table 2.1**:

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

Event	Probable class which performs logging (interface names are in brackets)
<p>User service submission arrives and is authorized (includes all service and corresponding reservation parameters in log):</p> <pre>[11:43:37] ===== NEW SERVICE REQUEST ===== [11:43:37] Name: test_user_1, home domain: http://test_host_A:8080/jra3/services/Interdomain , email: user@domain.edu [11:43:37] Justification: A test service request - Test Case 5 [11:43:37] Reservation: 0 ,desc: Reservation1 [11:43:37] Start port: portA.1 ,end port: portH.1 [11:43:37] Start time: Sat Jul 01 02:00:00 CEST 2006 ,end time: Thu Jul 06 01:59:59 CEST 2006 [11:43:37] Capacity: 800000000 ,delay: 17 [11:43:37] Bidirectional: true ,resilience: true</pre>	<p>AccessPoint (UserAccessPoint)</p>
<p>User service is submitted for schedule:</p> <pre>[11:43:37] Executing service: srv_0</pre>	<p>ServiceScheduler (ServiceListener)</p>
<p>Service state changed</p> <pre>[11:44:00] Service: srv_0 done with status scheduled</pre>	<p>Service</p>
<p>Reservation state changed</p> <pre>[11:44:00] Reservation: res_0 scheduled</pre>	<p>Reservation</p>
<p>Reservation processing steps, including:</p> <p>current path under analysis (include path), eg:</p> <pre>[11:43:37] Reservation [res_0]: trying path { [PortA.1, Port1.1], [Port1.5, Port3.1], [Port3.3, Port4.1], [Port4.2, Port5.5], [Port5.4, Port6.3], [Port6.4, Port8.1], [Port8.2, PortH.1], }</pre>	<p>Reservation</p>

<p>inter-domain constraints check (report failures only), eg:</p> <pre>[12:57:16] Reservation [res 0]: Acquired path: {[PortA.1, Port1.1], [Port1.5, Port3.1], [Port3.3, Port4.1], [Port4.2, Port5.5], [Port5.4, Port6.3], [Port6.4, Port8.1], [Port8.2, PortH.1],}</pre> <p>[12:57:16] delay too big!</p> <p>[12:57:16] vlan number cannot be agreeded!</p> <p>reservation request sent to neighbor (include neighbor identifier), eg:</p> <pre>[11:43:38] Schedule reservation: res_0 was sent to: https://test_host_3:8443/jra3/services/Interdomain</pre> <p>reservation schedule report arrives (include report results)</p> <pre>[11:44:00] Report schedule: res_0 arrived</pre> <pre>[11:44:00] Reserving resources: {resiliency=true} AND {cost=10.0, delay=3.0} AND {vlans=[200, 200], }</pre> <p>or</p> <pre>[11:43:52] Reservation [res 0]: PATH FAILURE: {[PortA.1, Port1.1], [Port1.5, Port3.1], [Port3.3, Port4.1], [Port4.2, Port5.5], [Port5.4, Port6.3], [Port6.4, Port8.1], [Port8.2, PortH.1],} failed, msg: 'Global constraints not fulfilled'</pre> <p>any failure/refuses (include reasons if available)</p>	
--	--

Table 2.1: Log events

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

3 Behavioural model – sequence diagrams

This section contains sequence diagrams that provide the time variant relationships between classes/objects. The call arrows on diagrams are associated with method names.

3.1 Sequence diagram for user's request submission

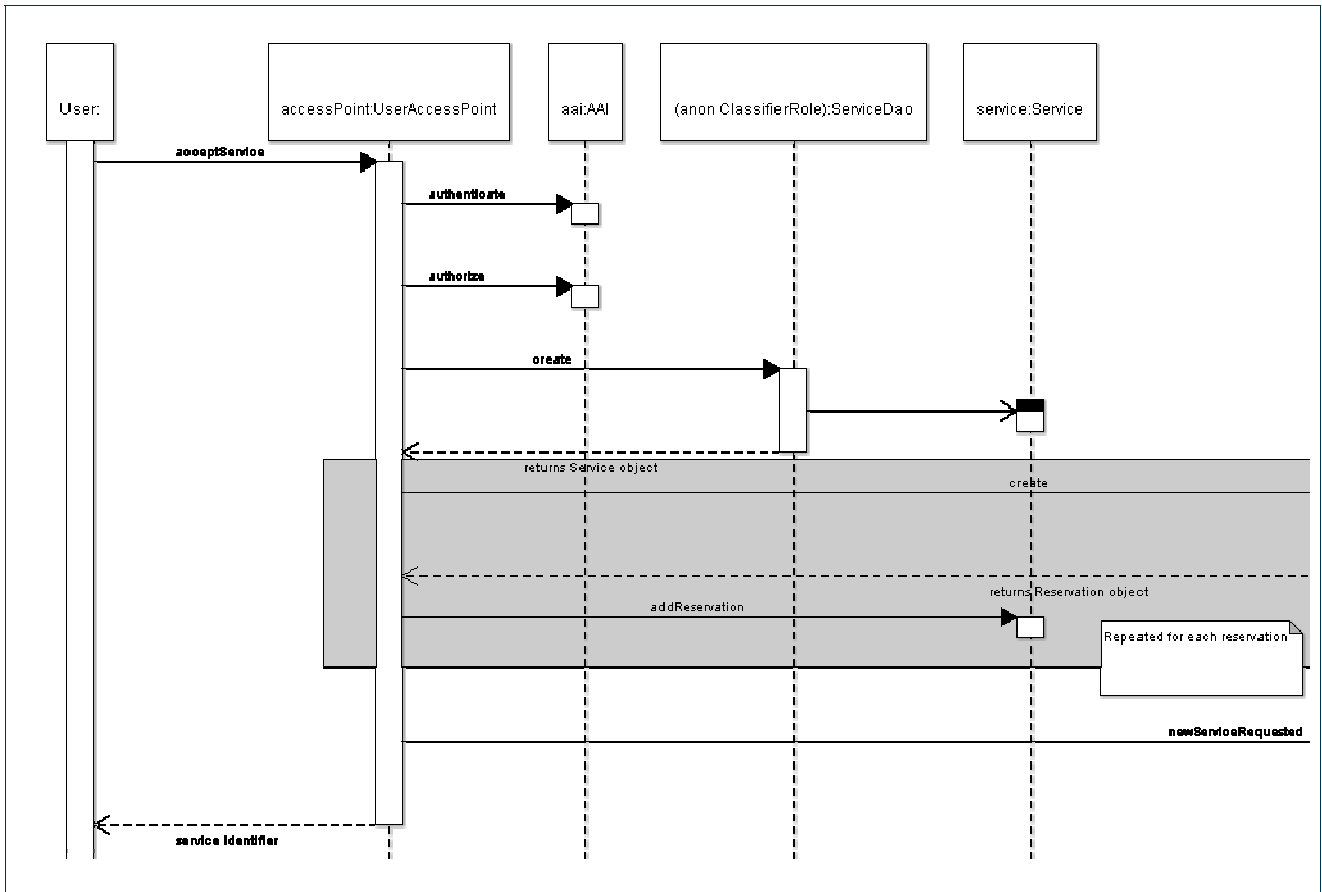


Figure 3.1: Sequence diagram for user's request submission (Part A)

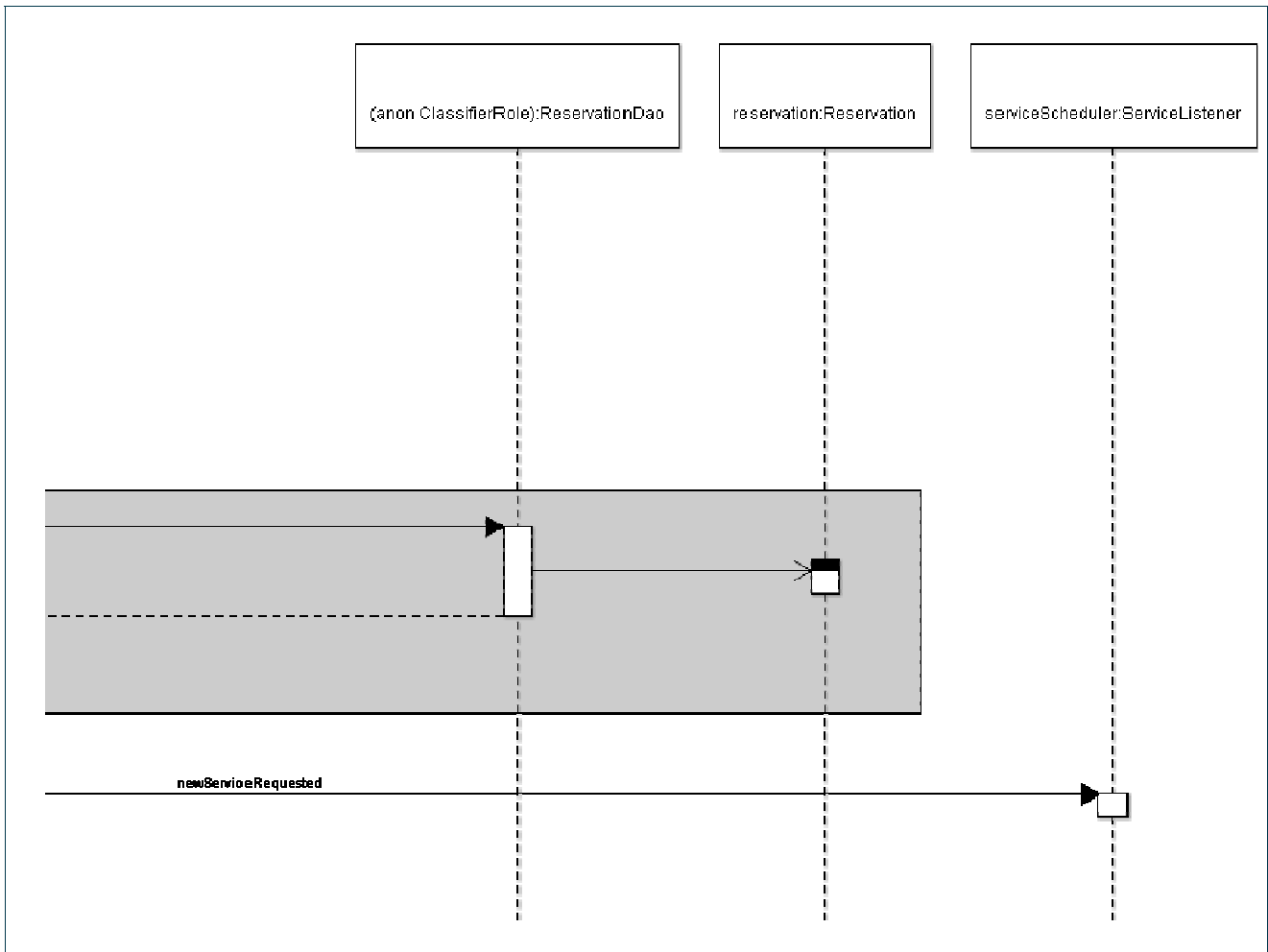


Figure 3.2. Sequence diagram for user's request submission (Part B)

Figure 3.1 and **Figure 3.2** present the execution flow for accepting a user's request. Through the system WebService, a user invokes the `acceptService()` method, including all service and reservation parameters. The user and the message are then authenticated and authorised with the `aaI` class. If the operation succeeds, internal representations of service and reservations are created with usage of DAO objects. Since the `Service` object is ready, through `newServiceRequested()` method, it is forwarded to the `serviceScheduler` object, which is now responsible for proper service processing. The user is notified about a successful operation with an accepted service identifier.

3.2 Sequence diagram for request status check

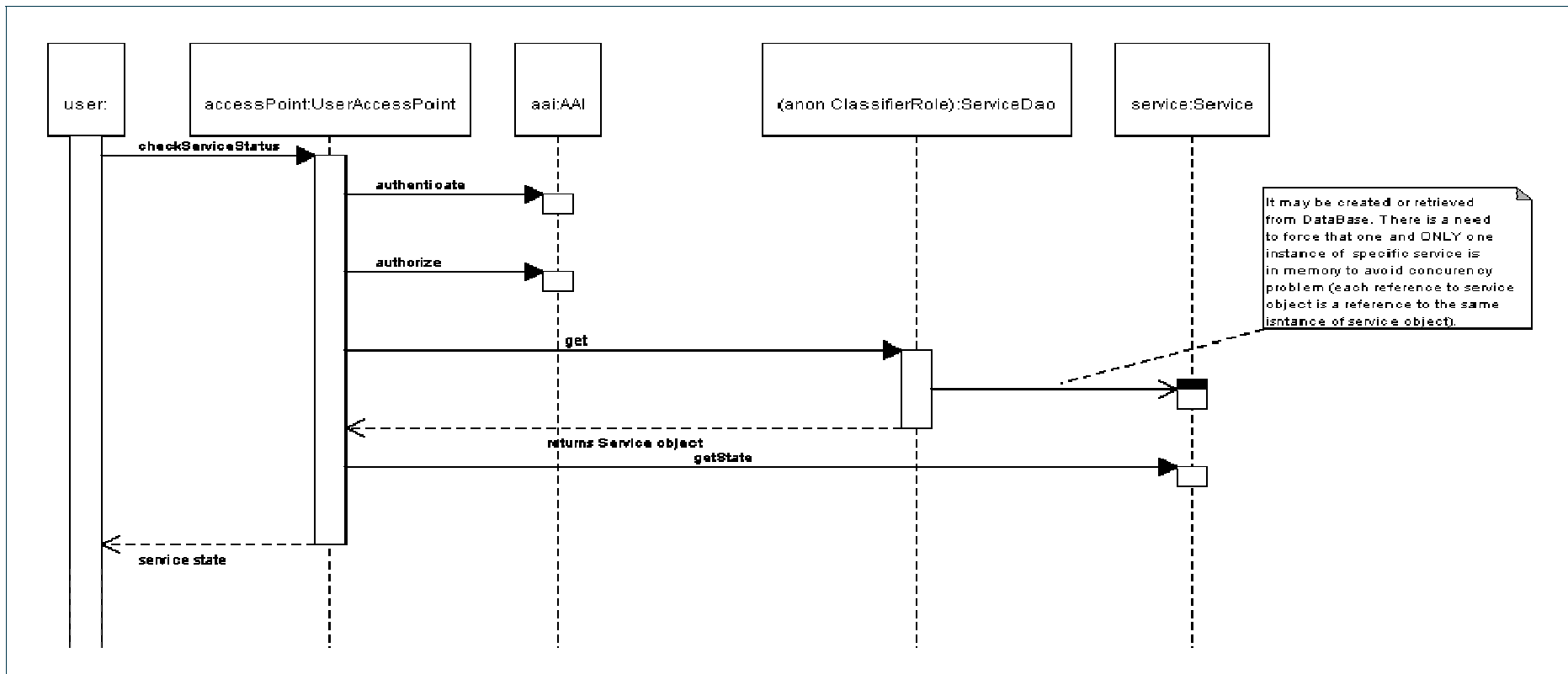


Figure 3.3: Sequence diagram for request status check

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

Since the service is accepted and awaits for execution, the user can check its status. The processing flow for this operation is depicted on **Figure 3.3**. The User accesses the system with the `checkServiceStatus()` method through the WebService interface. The `accessPoint` object uses the `aai` object to authorize and authenticate the request. If those operations are done with success, the proper representation of the `Service` object is retrieved through the `ServiceDao` class, using the identifier specified by the user. Once proper service is found, its status is taken by the `getState()` method, and the results are returned to the user as a response to the WebService request.

3.3 Sequence diagram for user's service cancel request

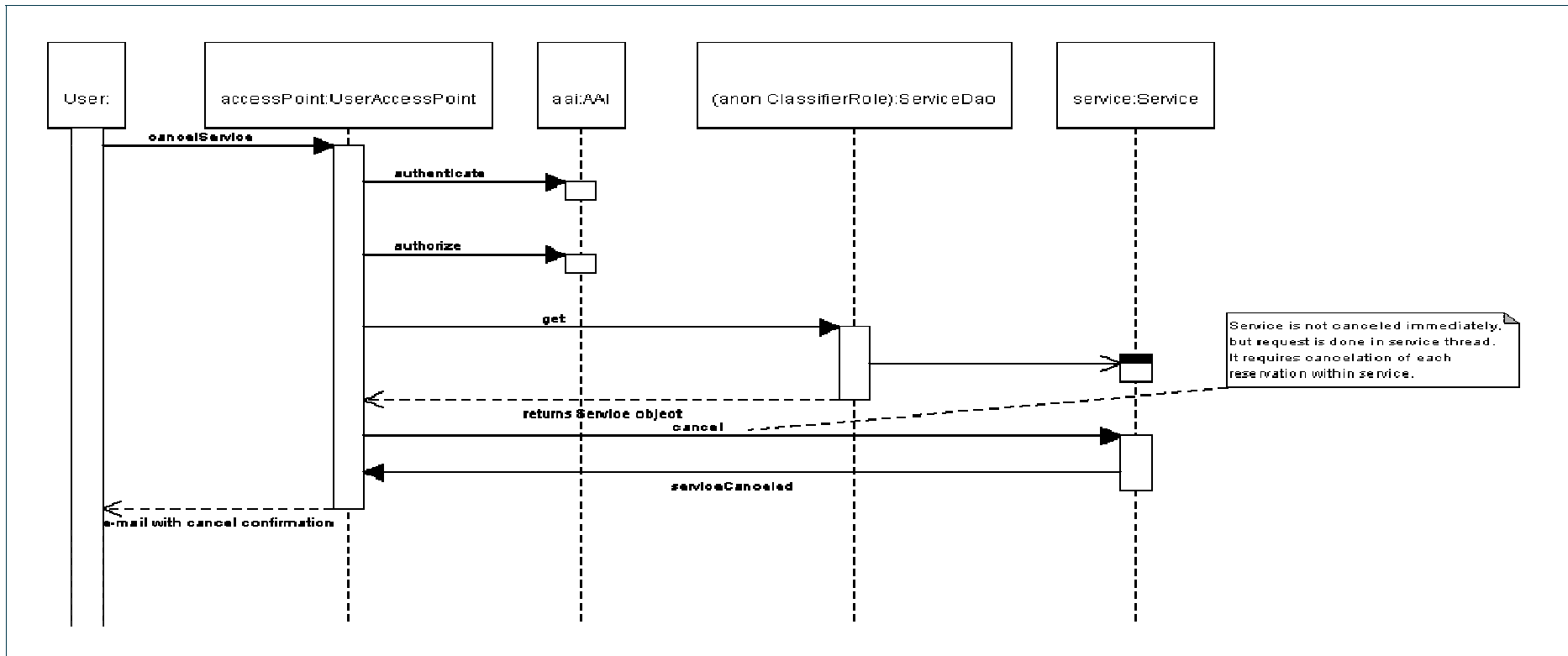


Figure 3.4: Sequence diagram for user's service cancel request

The user may cancel his request at any time. **Figure 3.4** shows the processing flow for the reservation's cancellation. The flow is very similar to the status check, except, that after the proper *Service* object is found, the `cancel()` method is invoked, instead of the `getState()` method. Service cancellation is a complex issue and requires some processing and inter-domain communication. Therefore the user is notified about the successful cancellation with an e-mail after the service is permanently removed from the system.

3.4 Sequence diagram for service schedule processing

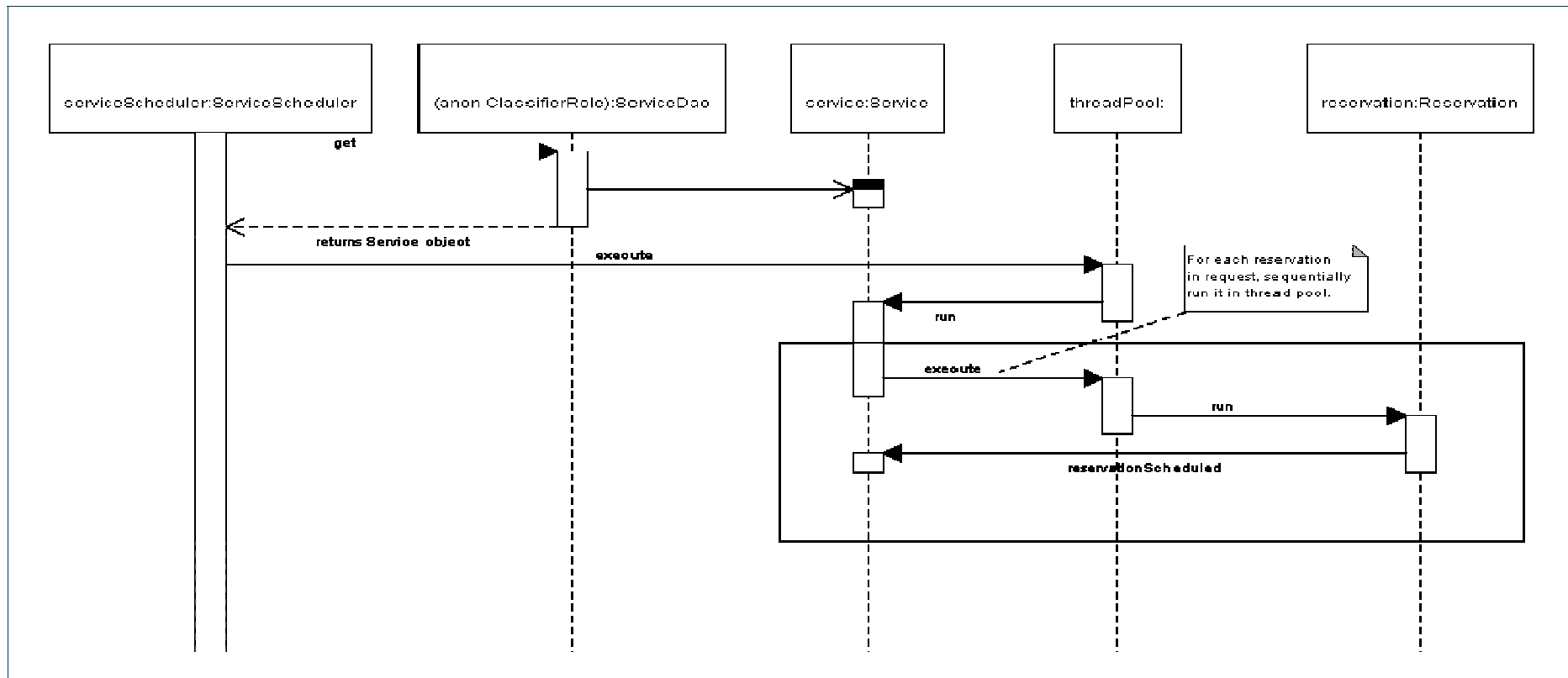


Figure 3.5: Sequence diagram for service schedule processing

Services are awaiting for execution queued in the `serviceScheduler` object (see **Figure 3.1** and **Figure 3.2**). If the system decides to start a reservation process, it takes the first service request from the queue, and then users `ServiceDao` class to find the proper instance of `Service` to execute (**Figure 3.5**). A dedicated thread manager `threadPool` is then asked with the `execute()` method to perform the service booking operations. Then the `Service` instance executes each `Reservation` as a separate thread (again with assist of the `threadPool` object). Only one reservation from a single service is executed at the same time in one domain. Only one service may be executed at the same time in one domain. It is possible to perform more than one reservation at the same time in a single domain, only if those reservations are requested by the neighbour BoD systems.

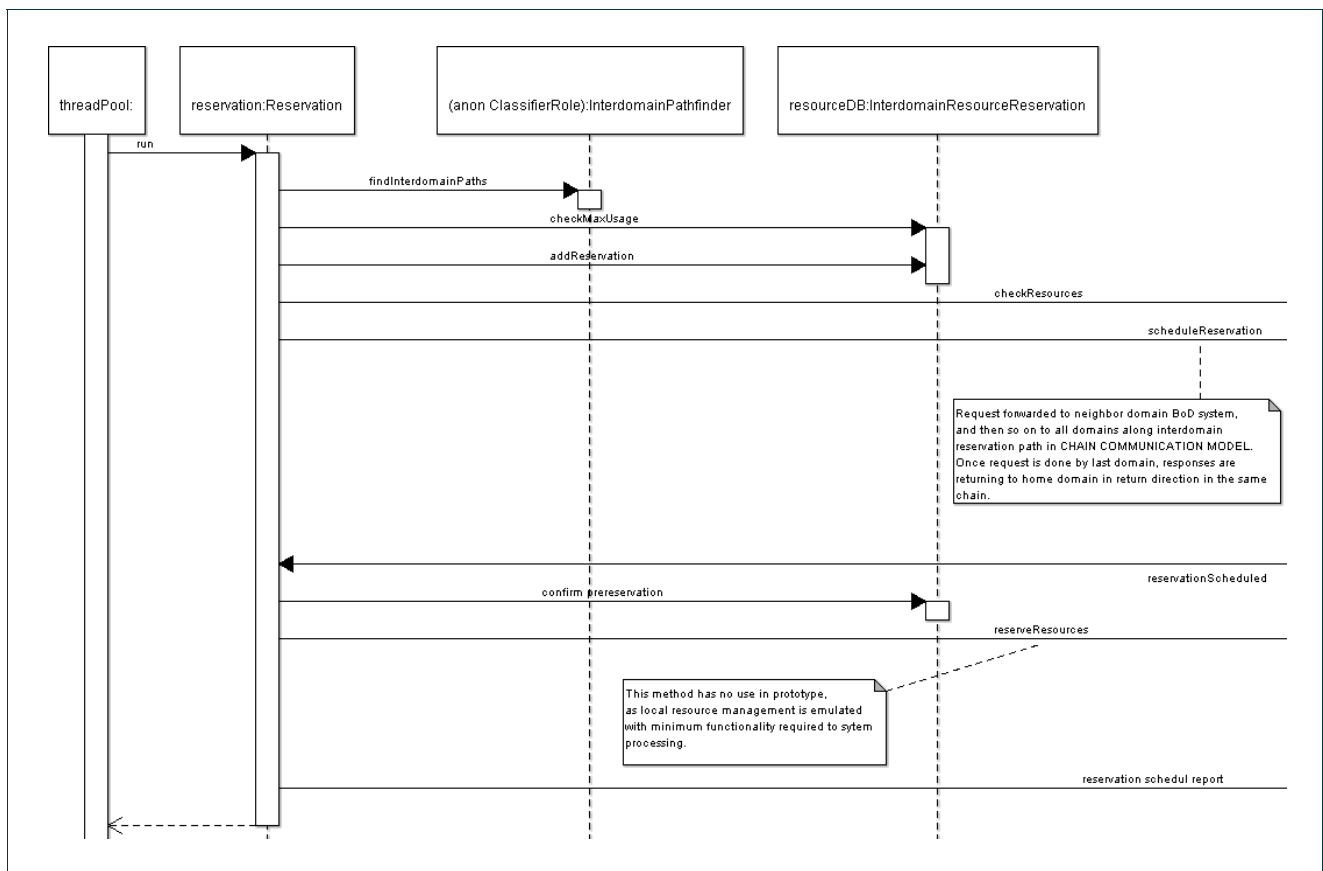


Figure 3.6: Sequence diagram for reservation processing in home domain (Part A)

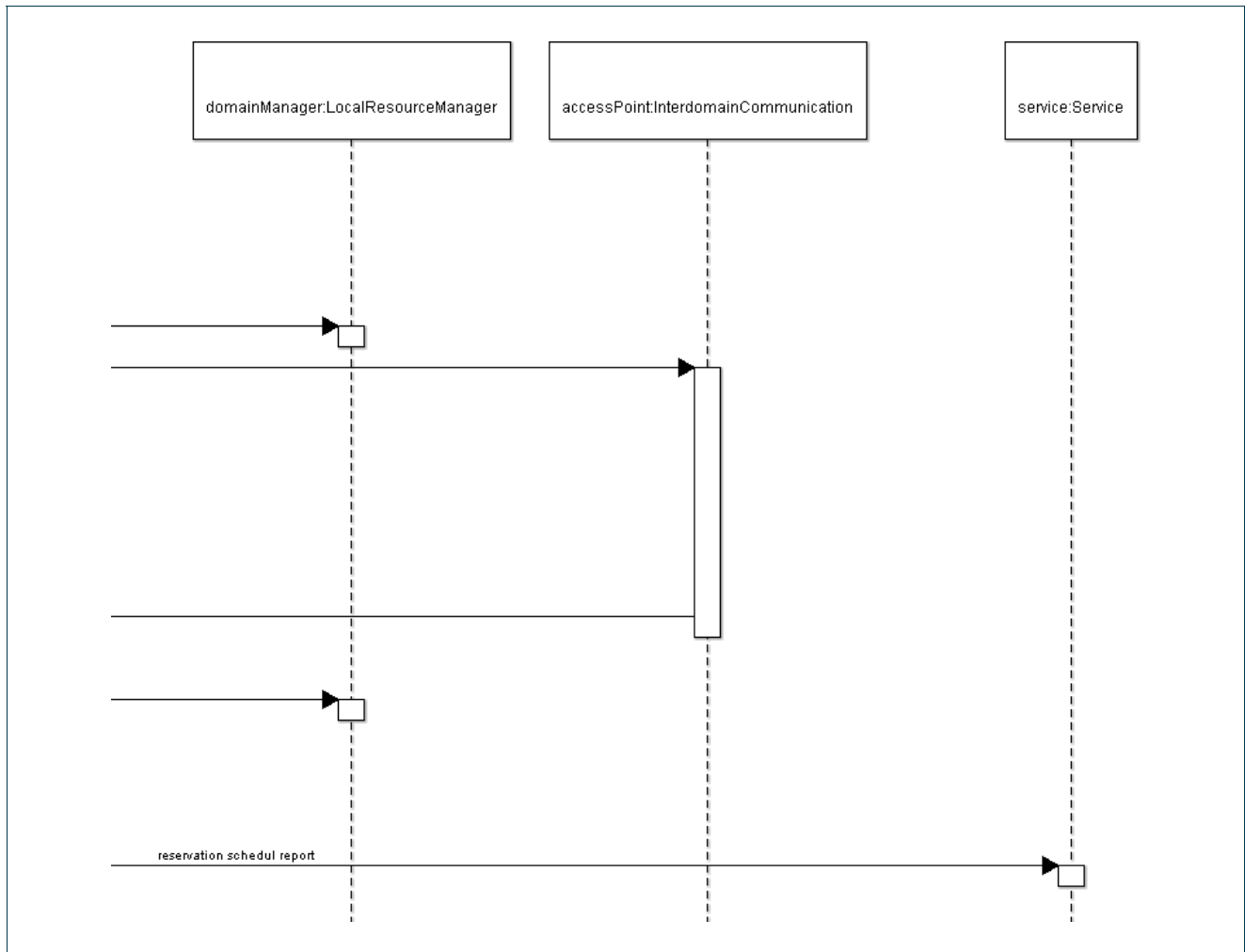


Figure 3.7: Sequence diagram for reservation processing in home domain (Part B)

Figure 3.6 and Figure 3.7 show the processing flow for the single reservation execution in the home domain. At the very beginning the Reservation object searches for inter-domain paths with InterdomainPathfinder class. Then the InterdomainReservationResources class is asked to check, if there are enough resources for this reservation in the current domain (for inter-domain link only). If there is enough resources, a pre-reservation is performed at the inter-domain level. Then the domainManager object is asked, if the request can be scheduled at the intra-domain level and what conditions should be meet. Having this information, the reservation request is sent to the next domain on the reservation path, through the accessPoint object. Once the response has arrived and is successful (with the reservationScheduled() method), the pre-reservation is confirmed at the InterdomainReservationResources object. The domainManager object is ordered to make the intra-domain reservation. Since resources are booked, the Service is notified that this reservation is done (scheduled).

If a neighbor domain reports a reservation failure (the reservationFailure() method instead of the reservationScheduled() method), then the system takes another inter-domain reservation path provided by the InterdomainPathfinder class and executes all the steps again.

Project:	GN2
Deliverable Number:	DJ3.4.1
Date of Issue:	11/09/06
EC Contract No.:	511082
Document Code:	GN2-06-184v4

3.5 Sequence diagram for reservation processing in domain somewhere along reservation path

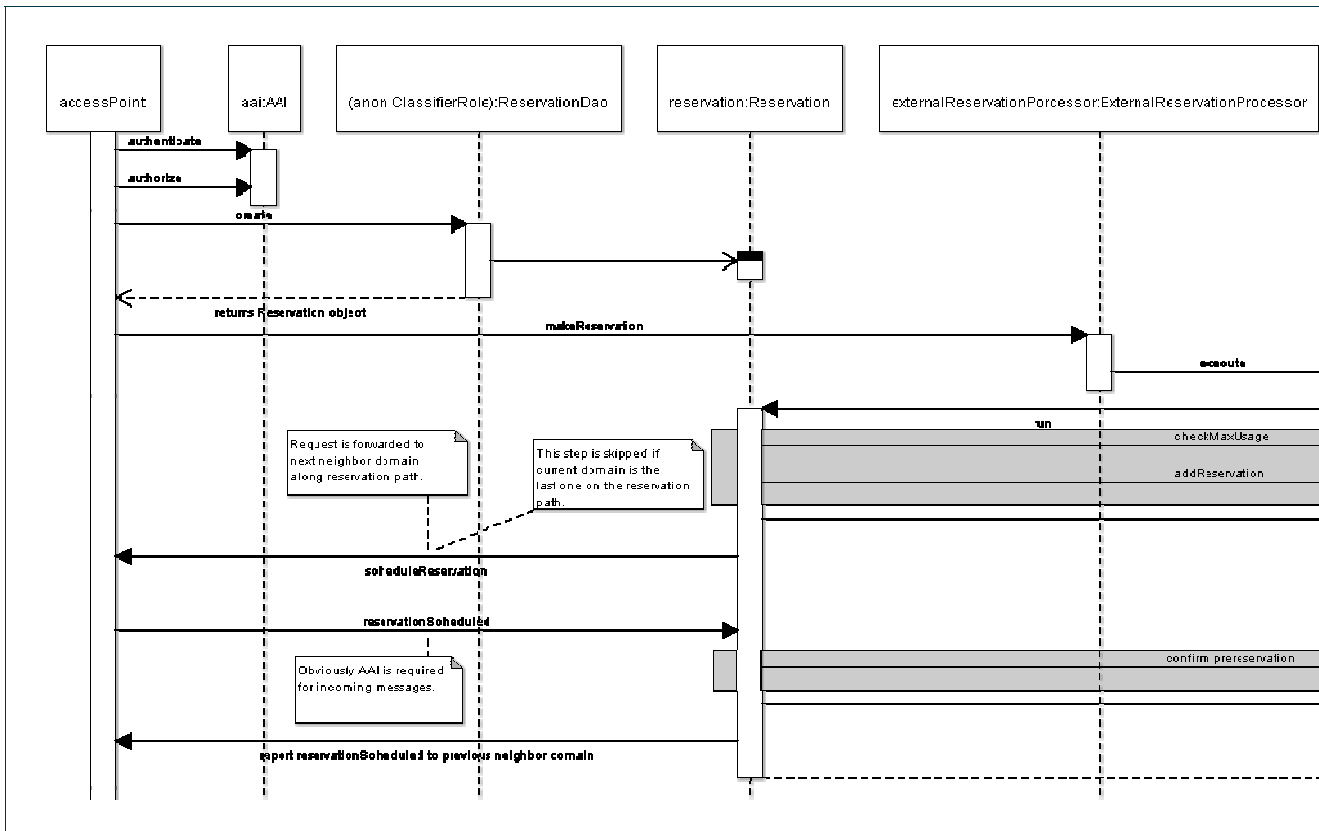


Figure 3.8: Sequence diagram for reservation processing in domain somewhere along reservation path (Part A)

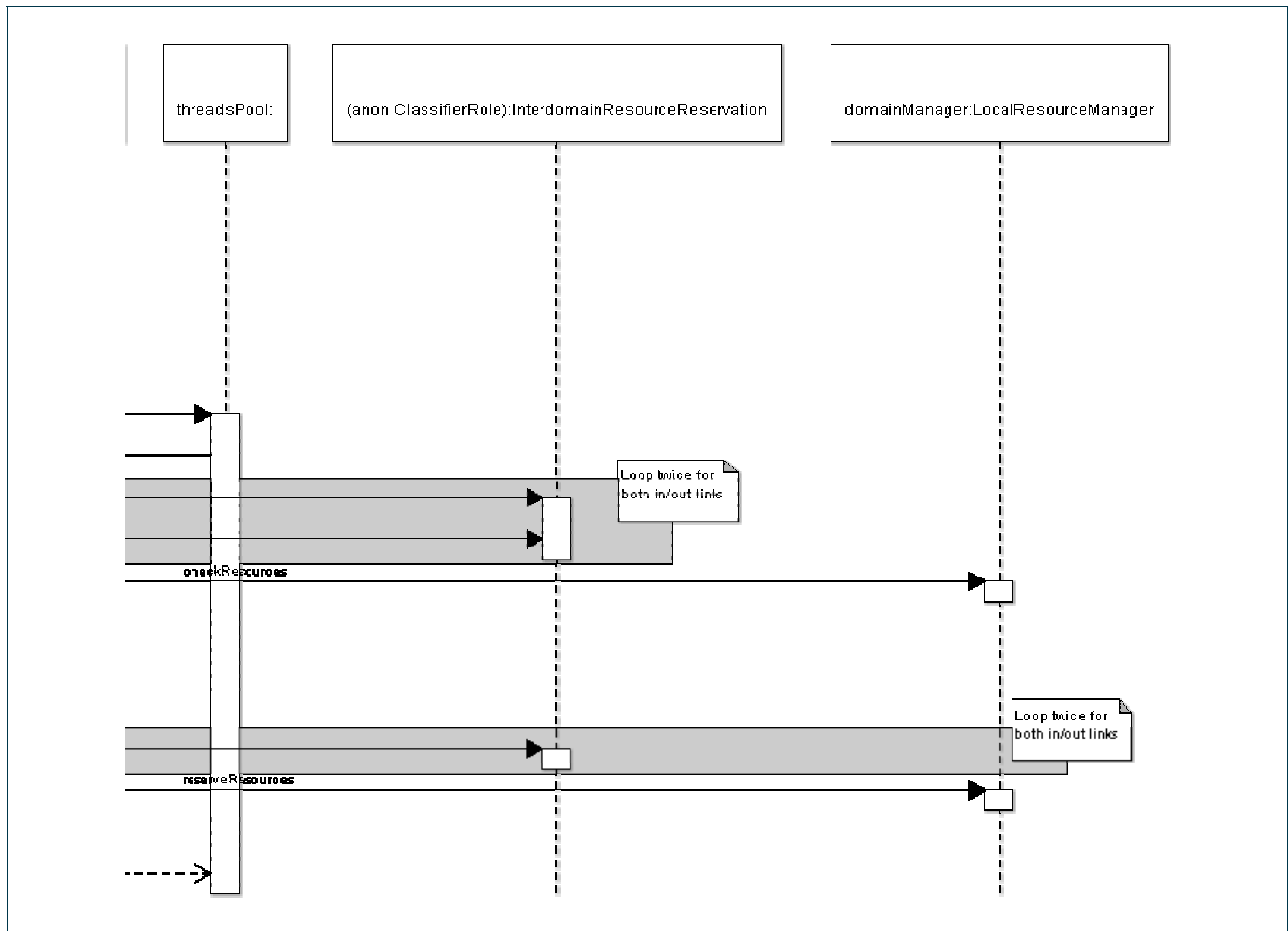


Figure 3.9: Sequence diagram for reservation processing in domain somewhere along reservation path (Part B)

Figure 3.8 and **Figure 3.9** show the single reservation processing in any domain in the middle of a reservation path. After authentication and authorization of the arriving reservation request, the system creates the proper representation of the `Reservation` object for internal use (using the `ReservationDao` class). The reservation is executed through the `externalReservationProcess` class and the `threadPool` thread controller object. The reservation checks availability of domain ingress and egress links with the `InterdomainResourceReservation` class (similar as in the home domain) and makes the pre-reservation. Then intra-domain conditions are checked in the `domainManager` class. If everything succeeds, the request is forwarded to the next domain on the reservation path. Once the response arrives, the pre-reservations are confirmed and information about the reservation schedule is sent to the domain originating the request.

If the current domain is the last on the reservation path, the successful pre-reservation is automatically confirmed and the schedule information message is sent to the domain where the request originated from.

4 Conclusions

The current design of the IDM prototype assumes some limitations to keep the implementation effort at reasonable levels while at the same time enable the observation of the system behaviour. Further system development, particularly the implementation of the IDM Phase 1 release, will require some redesign and analysis of the chosen assumptions. The interface for AA functionality is probably insufficient to introduce the eduGAIN JRA5 functionality in the JRA3 BoD system. Also the pathfinder and DM are implemented as “simulators” of the corresponding modules’ behaviour. The interfaces to these two components are designed as final, but the prototype testing may provide additional requirements to them. The network abstraction is another issue to tackle in the next step of development, as it does not fully reflect the abstract representation required by the system to run. The prototype was not equipped with any database engine and therefore there was no base for implementation of a any transaction mechanism. All information is stored in volatile memory, thus a system failure may turn some reservations and booked resources inconsistent. As the main goal of the prototype was to track the processing flow and behaviour of the system in regular operating conditions, the number of handled errors was limited to minimum. The communication problems between domains and modules should be considered in detail during the next implementation phases.

Despite the limitations mentioned, the IDM prototype functionality is sufficient to evaluate the system’s performance and find potential bottlenecks. This includes also some unresolved issues and assumptions that should be clarified during prototype tests, showing critical system components which may require redesigning. Also most of the already existing code can be used for building the final system, so the next IDM implementation phase can be finalized in shorter time.

5 References

- [GN2DJ331] GEANT2 Deliverable DJ3.3.1: GÉANT2 Bandwidth on Demand Framework and General Architecture, 20 Dec 2005, http://www.geant2.net/upload/pdf/GN2-05-208v7_DJ3-3-1_GEANT2_Initial_Bandwidth_on_Demand_Framework_and_Architecture.pdf
- [GN2DJ332] GEANT2 Deliverable DJ3.3.2 Functional Specification of GÉANT2 Inter-domain Manager (IDM) Prototype, 9 June 2006, http://intranet.geant2.net/upload/pdf/GN2-06-091v3-DJ3.3.2_FunctionalSpecificationGEANT2_IDM_Prototype.pdf,

6 Acronyms

AAA	Authorization, Authentication and Accounting
AAI	Authorization and Authentication Infrastructure
BoD	Bandwidth on Demand
DB	Database
DM	Domain Manager
IDM	Inter Domain Manager
JVM	Java Virtual Machine
PF	Path Finder (module)
URL	Uniform Resource Locator