Diligent

A DIgital Library Infrastructure on Grid ENabled Technology

Deliverable No D1.2.2:

"DL Creation & Management services specification report"

September 2005

Document Information

Project	
Project Title:	DILIGENT, A **DI**gital **L**ibrary **I**nfrastructure on **G**rid **EN**abled Technology
Project Start:	1st Sep 2004
Call/Instrument:	FP6-2003-IST-2/IP
Contract Number:	004260
Document	
Deliverable number:	D1.2.2
Deliverable title:	DL Creation & Management services specification report
Contractual Date of Delivery:	Month 11
Actual Date of Delivery:	15th September 2005
Editor(s):	CNR – ISTI
Author(s):	H. Avancini, L. Candela, P. Fabriani, P. Pagano, P. Roccetti, M. Simi
Reviewer(s):	C. Langguth, H. Schuldt (UMIT)
Participant(s):	CNR – ISTI, ENG
Workpackage:	WP1.2
Workpackage title:	DL Creation & Management
Workpackage leader:	CNR - ISTI
Workpackage participants:	CNR – ISTI, UoA, CERN, ENG
Est. Person-months:	19
Distribution:	Confidential
Nature:	Report
Version/Revision:	1.0
Draft/Final	Final
Total number of pages: (including cover)	156
File name:	D1.2.2-FinalVersion.doc/.pdf
Key words:	*Digital libraries, UML, Services Architecture, Services Design*

Disclaimer

This document contains description of the DILIGENT project findings, work and products. Certain parts of it might be under partner Intellectual Property Right (IPR) rules so, prior to using its content please contact the consortium head for approval.

In case you believe that this document harms in any way IPR held by you as a person or as a representative of an entity, please do notify us immediately.

The authors of this document have taken any available measure in order for its content to be accurate, consistent and lawful. However, neither the project consortium as a whole nor the individual partners that implicitly or explicitly participated the creation and publication of this document hold any sort of responsibility that might occur as a result of using its content.

This publication has been produced with the assistance of the European Union. The content of this publication is the sole responsibility of DILIGENT consortium and can in no way be taken to reflect the views of the European Union.

The European Union is established in accordance with the Treaty on European Union (Maastricht). There are currently 25 Member States of the Union. It is based on the European Communities and the member states cooperation in the fields of Common Foreign and Security Policy and Justice and Home Affairs. The five main institutions of the European Union are the European Parliament, the Council of Ministers, the European Commission, the Court of Justice and the Court of Auditors. (http://europa.eu.int/)

DILIGENT is a project partially funded by the European Union

Table of Contents

Table of Figures

Summary

This report presents the result of the activity conducted in the various design tasks (*T1.2.1.a Information Service design*, *T1.2.2.a Broker & Matchmaker Service design, T1.2.3.a Keeper Service design, T1.2.4.a Dynamic VO Support Service design*, and *T1.2.5.a VDL Generator Service design*) of the *WP1.2 DL Creation & Management* of the DILIGENT project during the period February 1st - August 31st 2005. It completes the service specification activity of the DILIGENT Collective Layer and replaces the interim service specification presented with the D1.2.1 DL Creation & Management Services Specification interim report.

The D1.2.2 DL Creation & Management services specification report, by relying on the functions and features of the DILIGENT system, as reported in *D1.1.1 Test-bed functional specification* [1],

i) identifies the functionalities related with the services forming this functional area,

ii) specifies the functionalities offered by each service, and

iii) presents the service specification in order to capture and convey on the most significant architectural decisions that have been made by the services.

Executive Summary

The objective of the *D1.2.2 DL Creation & Management services specification report* is to complete the specification of the DILIGENT Collective Layer, that is composed by the Information Service, the Broker & Matchmaker, the Keeper, and the Dynamic VO Support, plus the VDL Generator Service. The complete design phase of these services is spread out over 12 months and it is organized into three sub-phases that started guiding the design of the DILIGENT services belonging to the other layers and allowing to initiate an early implementation of the Collective Layer services aimed to test the design choices, and finally end providing all details required during the services implementation, testing, and maintenance phases. In order to achieve this goal three reports, each adding further level of detail to the specification, have been planned. These reports are respectively i) the D1.2.1 interim report, ii) this report, the *D1.2.2 DL Creation & Management services specification report*, and finally iii) the *D1.2.3 DL Creation & Management services detailed design report* due in January 2006.

The entire specification activity is conducted according to the guidelines of the Unified Process methodology. In this context, the Unified Modeling Language is used to formalize the various elements of the specification and their relationships needed to model the DL Creation & Management services.

The D1.2.1 interim report has already identified the subset of the functionalities reported in the *D1.1.1 Test-bed functional specification* that has to be provided by the Collective Layer services. In this report, the specification activity clarifies the constraints imposed by the environment in which the system/software must operate, e.g. the WS Resource Framework, the need to reuse existing assets, e.g. the Aggregator Framework, and the imposition of various standards that have been taken into account.

Having fixed these aspects, a set of UML views are used to present: the architecture of each service from the functional, logical, and deployment point of view; the relationships with the other services; the dependencies among the service and the environment in which the software must operate; the need and the approach to reuse existing assets. During this activity all the objects required by the system have been identified and described. In particular:

a) The Use-Case View has been used to present use cases that represent significant functionality of the service, that have a large architectural coverage, and that illustrate a delicate point of the architecture;
b) The Logical View has been used to describe the architecturally significant parts of the service design model, such as its decomposition into subsystems and packages. Moreover, for each significant package, its decomposition into classes and class utilities is also presented;
c) The Deployment View has been used to illustrate one or more physical network configurations on which the service is deployed and run to emphasize the level of distribution of the component parts of each service.

After this overview a detailed section has been produced for each service illustrating:

a) The design considerations, i.e. the issues which need to be addressed or resolved before attempting to devise a complete design solution.
b) The service decomposition in components, each of which is explained through the state, operations, profile, status, dependencies, and requirements description.

Finally, taking into account that this report has been conceived as the second step of the design activity, spread out over three reports, a set of labels has been used to improve its readability. These labels, with the associated semantics, are reported below:

a) *[to be detailed in D1.2.3]*. The information reported in the section are the result of the initial specification phase and more detail information will be added in the Detailed design (D1.2.3). This information is related to the mapping of the specification classes to the design classes. Specification classes represent roles played by instances of design elements; these roles may be fulfilled by one or more design model elements. In addition, a single design element may fulfil multiple roles.

   In both cases the semantics related to the boundary classes, the interfaces, the library, and the identified services would be preserved.

b) [*to be provided in D1.2.3*]. The service specification requires an in depth investigation. The investigation activity is already started and will also be influenced by the results of the early implementation made to support the experimentation prototype.

1 INTRODUCTION

The whole DILIGENT system engineering is conducted according to the guidelines of the Unified Process methodology. Following this methodology the architecture of a complex software system is the organization or structure of the system's significant components interacting, through interfaces, with components composed of successively smaller components and interfaces.

The first decomposition of the system into components has already been made in the Description of Work document, where the main services (or classes of services) were identified and used to structure and organize the technical WPs. As a consequence, the design phase will be conducted in parallel over all the technical WPs responsible for the identified DILIGENT main services.

This report presents the result of the activity conducted within the design tasks of the *WP1.2 DL Creation & Management*, i.e. *T1.2.1.a Information Service design*, *T1.2.2.a Broker & Matchmaker Service design, T1.2.3.a Keeper Service design, T1.2.4.a Dynamic VO Support Service design*, and *T1.2.5.a VDL Generator Service design*. The design phase of the WP1.2 spreads out over 12 months and is further organized into three sub-phases whose final goal is to produce a detailed service specification that will guide the subsequent services implementation phase. The final goal will be achieved by producing three reports, each adding further details to their services specification. These reports are respectively i) the interim report D1.2.1, ii) the *D1.2.2 DL Creation & Management services specification report* (this report), and finally iii) the *D1.2.3 DL Creation & Management services detailed design report* due in January 2006.

The services belonging to WP1.2, objective of this report, jointly with those provided by the gLite Grid middleware released by the EGEE project, manage the resources and applications needed to run distributed digital libraries as service oriented applications.

Figure 1 shows these services and the main interactions among them. In particular, the functionalities provided by the DL Creation & Management are:

1. the monitoring and discovering of all the available DILIGENT resources (Information Service)
2. the creation of the trusted environment needed for ensuring a controlled sharing of these resources (Dynamic VO Support Service)
3. the implementation of a global strategy offering the optimal use of the resources supplied by the DILIGENT infrastructure (Broker & Matchmaker Service)
4. the orchestration needed to maintain up and running the pool of resources that populate the various DLs and to ensure certain levels of fault tolerance and QoS (Keeper Service)
5. the support for users that want to define their own DLs (VDL Generator Service).

These five services are high level services (or conceptual services). They are decomposed in one or more *components* that can be WSRF-compliant services, portlets, libraries, and sub-systems as reported in the Deployment View and Service Details sections.

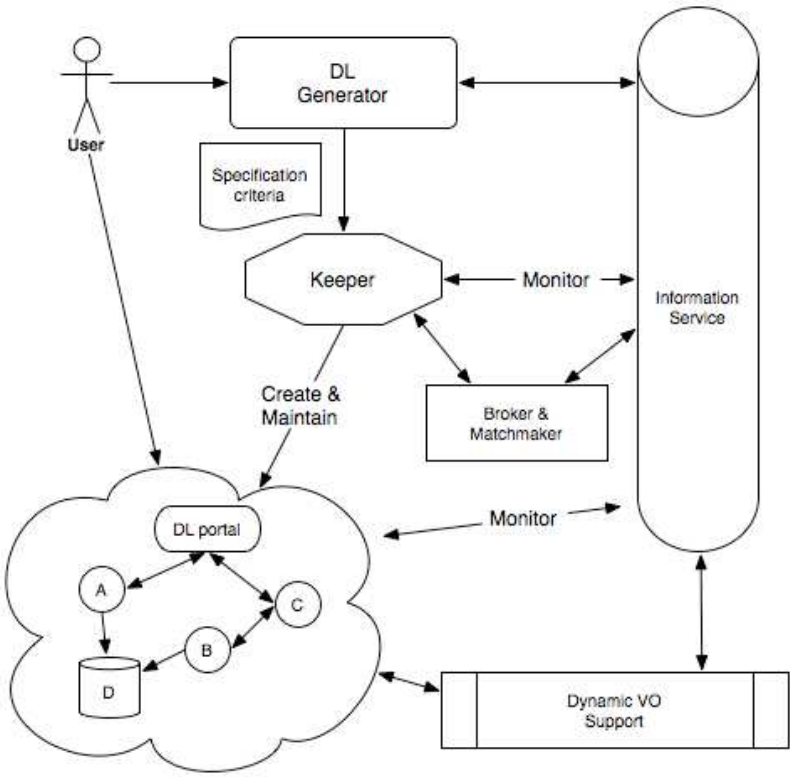

Figure 1. DL Creation and Management services

The outline of this report is as follows: the next section presents the rationale of the services specification report illustrating the layout and the rules proposed in producing the service specifications of each of the cited services; Section 3 reports the DILIGENT Resource Model, i.e. it describes, via a set of UML class diagrams, the most important characteristics and attributes of the resources forming the DILIGENT system; Section 4 introduces the Information Service specification; Section 5 presents the Broker & Matchmaker Service specification; Section 6 reports the Keeper Service specification; Section 7 describes the Dynamic VO Support Service specification; Section 8 introduces the VDL Generator Service specification; Section 9 concludes reporting the follow up of this report, in particular its relationships with the future *D1.2.3 DL Creation & Management services detailed design*; and finally Appendix A reports guidelines in designing DILIGENT service security.

2 RATIONALE OF SERVICES SPECIFICATION REPORT

The service specification is one of the steps of the design phase. In particular, the design phase, based on the functional view reported into the *D1.1.1 Test-bed functional specification*, is in charge to identify and define data elements, components, interfaces, outputs and whatever is needed for a rapid prototyping of all the services. The first specification of these characteristics has been supplied via the services specification interim reports set up for each of the WP responsible for supplying DILIGENT Services, i.e. WP1.2 – WP1.6.

The D1.2.2 DL Creation & Management Services Specification report includes the design considerations (i.e. the issues which need to be addressed or resolved before attempting to devise a complete design solution), and the services decomposition in components explained through the state, operations, profile, status, dependencies, and requirements description of each of them.

2.1 Methodology

The first step in designing each single service has been to identify the subset of the functionalities reported in the *D1.1.1 Test-bed functional specification* that has to be provided by the WP1.2 services. This task is simplified thanks to the process adopted during the realization of the functional specification. In that process each DILIGENT partner i) participates in the analysis phase by understanding the specific aspects it is involved and has more expertise in, and ii) contributes to the modelling of the identified functionality into the whole functional picture.

Having identified the functionalities of competence of each service, each designer must take into account that the architectural shape is influenced also by other characteristics: there are constraints imposed by the environment in which the system/software must operate; by the need to reuse existing assets; by the imposition of various standards; by the need for compatibility with existing systems, and so on. Some of these characteristics can be service specific but there are constraints that are global in nature, i.e. they influence all the services forming the system. Among those global constraints are the following three that must be carefully considered:

- The DILIGENT system is based on a *Service Oriented Architecture (SOA)*;
- The DILIGENT system will make use of the resources offered by the EGEE infrastructure;
- The DILIGENT system must exploit the capabilities provided by the gLite middleware.

Having fixed these aspects, another point to clarify is related with *components* and *services identification*. It is important to emphasize that each main service, e.g. the Keeper, can be designed on its own and follows the most appropriate architectural pattern [2] even if it must be compliant with the DILIGENT system set of constraints.

One of the most important design choices is on identifying the component services of each single main service. Many early adopters of SOA and Web services realized quickly that the proliferation of Web services does not make for a sound SOA model. Service identification consists of a combination of *top-down, bottom-up, and middle-out* techniques of domain decomposition, existing asset analysis, and goal-service modelling [3]. In the top-down view, the specification for services is provided. It consists of the decomposition of the domain into its functional areas and subsystems, including its flow or process decomposition into processes, sub-processes, and high-level use cases. This activity is partially covered by the DoW and the D1.1.1 "Test-bed Functional Specification", but needs to be reiterated by

each single main service to have fine-grained service decomposition. In the bottom-up portion of the process, existing systems are analyzed and selected as viable candidates for providing lower cost solutions to the implementation of service functionalities. In this process, it is necessary to analyze and leverage API's, transactions, and modules from legacy and packaged applications. In some cases, componentization of the legacy systems is also needed to re-modularize the existing assets for supporting service functionality. This activity is particularly needed in the DILIGENT project that is partially built by integrating different technologies. The middle-out view consists of goal-service modelling to validate and discover other services not captured by either top-down or bottom-up service identification approaches.

As a consequence, the picture of the system presented in the DoW is just a course-grained overview of the system, while the real architecture of each DILIGENT Service is expressed in this document in terms of Web services, sub-systems and components constituting it.

DILIGENT project uses the Unified Modelling Language to represent in a formal mode:

- The architecture of each service;
- The relationships with other services;
- The dependencies among the service and the environment in which the system/software must operate;
- The need and the approach to reuse existing assets;
- The imposition of various standards and protocols.

In order to reach the goal of the services specification report the layout proposed in the following section has been adopted. This layout uses a section reporting service specification for each of the services involved by the usage of different architectural views.

2.2 The Services' Section Layout

The goal of this section is to provide an architectural overview of the service. It is also intended to capture and convey the significant architectural decisions that have been made on the service. After an introduction reporting a description of the service as well as an indication of the system functionalities covered, three further sections provide a quick glance of the service. These three sections present the service from three different points of view:

1. Use-case View
2. Logical View
3. Deployment View

In order to use in a uniform way the UML formalism, it is important to remind that[1]:

- A **use case** is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor;
- A **package** is a general-purpose mechanism for organizing elements into groups;
- A **class** is a description of a set of objects that share the same attributes, operations, relationships, and semantics;
- A **component** is physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces;
- A **node** is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability.

[1] Taken from G. Booch, J. Rumbaugh, I. Jacobson "The Unified Modeling Language User Guide". Addison-Wesley.

- A **stereotype** is an extension of the vocabulary of UML, allowing to create new kinds of building blocks similar to existing ones but specific to a predefined problem.

Moreover, even if classes and components have many commonalities (e.g. both may realize a set of interfaces, both may have instances) there are also some important differences:

- Classes represent logical abstractions while components represent physical things that live in the world of bits;
- Components represent the physical packaging of otherwise logical components and are at a different level of abstraction;
- Classes may have attributes and operations directly. In general, components have operations that are reachable only through their interfaces.

In the following sections the contribution of each view is illustrated, together with the recommended stereotypes to use in the complete specification.

2.2.1 Use-Case View

This section presents use cases or scenarios from the use-case model that i) represent significant functionality of the service, or ii) have a large architectural coverage (i.e. exercise many architectural elements), or iii) illustrate a specific, delicate point of the architecture.

As a consequence the section contains a set of use cases, usually organized in packages, and for each use case a subsection containing:

- A brief description,
- Any significant description of the flow of events of the use case,
- Any significant description of relationships involving the use case (i.e. include, extend and communication),
- Any significant description of special requirements of the use case,
- Any significant picture of the user interface, if any, clarifying the use case.

The realization of these use cases will be described in the logical view section.

2.2.2 Logical View

This section describes the architecturally significant parts of the service design model, such as its decomposition into subsystems and packages. For each significant package, its decomposition into classes and class utilities is also presented. Moreover architecturally significant classes are introduced and described, in order to describe the responsibilities and to identify a few very important relationships, operations, and attributes. It also describes the most important use-case realizations.

The following stereotypes have been adopted for the classes: **boundary**, **control**, and **entity**. Apart from giving more specific process guidance, this stereotyping results in a robust object model because changes to the model tend to affect only a specific area. For instance, changes in the user interface will affect only boundary classes; changes in the control flow will affect only control classes, while changes in long-term information will affect only entity classes. However, these stereotypes are especially useful in helping designers to identify classes in analysis and early design. Just to give a quick overview:

- A **boundary class** is a class used to model interaction between the service's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the service presentation (such as the interface);
- A **control class** is a class used to model control behaviour specific to one or a few use cases. Control objects (instances of control classes) often control other objects,

so their behaviour is of the coordinating type. Control classes encapsulate use-case specific behaviour.

- An **entity class** is a class used to model information and associated behaviour that must be stored. Entity objects (instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or some real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the life of the system/service.

As a consequence the section mainly contains one or more Class Diagrams and their related brief descriptions. In particular, for each significant package will be included a subsection reporting:

- A brief description,
- For each significant class, a brief description and, if possible, a description of its major responsibilities, operations and attributes.

With respect to use-case realization the goal has been to illustrate how the system works, explaining how the various designed elements collaborate to achieve the functionality.

2.2.3 Deployment View

This section describes one or more physical network configurations on which the service is deployed and run. It can also describe the allocation of service elements and tasks to the physical nodes. A deployment diagram is used to better illustrate the configuration.

If there are many possible physical configurations, a typical one has been described and then general mapping rules to follow in defining others have also been reported.

It is important to stereotype the components using at least the following stereotypes:

- A **service component** represents a Web Service. It has an interface described in a machine-processable format (WSDL). Other services interact with it in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.
- A **library component**, i.e. a static or dynamic object and function library. These components are used to identify a bunch of code that can be shared and used by other components also;
- A **subsystem component**, i.e. any kind of system that can be treated as an independent system. It is usually entirely included in another component.
- A **portlet component**, i.e. a pluggable user interface components to be hosted by the portal engine capable of providing a presentation layer to the system.
- A **job component**, i.e. a component that may be executed on a Grid node, in particular on a Computing Element (CE).

Detailed information about any significant component has been presented by adding a subsection whose content is described in Section 2.2.4. In particular, it has been described in detail:

- Each service component that is needed to improve the definition of the boundary each service has. This allows to have a detailed description of the user interface and access methods each service offers;
- Each library component or subsystem component that may be or must be used also by other DILIGENT services.

2.2.4 Service design

2.2.4.1 Design considerations

This section describes the issues which have been addressed and resolved before attempting to devise the complete design solution of the service.

2.2.4.2 Components

A *component* of a service is "a part" of that service that can be hosted in different networked locations. In this section there is an explanation of the rationale that has led to the service decomposition explained in the subsequent sections. An overview of each component is also provided here.

Then, there are a number of sections that present the design of each component illustrating:

- The state description. Taking into account that the DILIGENT services are compliant with the WSRF specifications, this section lists and explains the WS-Resources managed, how they are created, which is their state, how their state is made persistent, etc.
- The operations. The list of the high level operations that can be performed by invoking the service are listed and described. In the next D1.2.3 detailed design report these operations will become one or more operations declared in the WSDL PortTypes of the service.
- The profile description. This section reports the description of the running instance profile that is stored in the DIS and that can be used to perform a profile based service discovery.
- The status description. The description of the status exported as WS-ResourceProperties [22] of the WS-Resources managed by the service are listed. It is stored in the DIS and it can be used to perform a status based service discovery.

Dependencies & Requirements. The description of the dependencies of the service with respect to the other DILIGENT services, the gLite services, and the other technologies are reported in this section.

2.2.4.3 Deployment scenario(s)

This final section of the service design reports a description of one or more deployment scenarios of the described components. It addresses questions like

- How many instances (may) exist per DL/VO?
- Are components replicated?
- In case of multiple instances, which is the distribution model? How are replicas synchronized?

3 DILIGENT RESOURCES MODEL

A DILIGENT resource is the basic component of any DL handled by the DILIGENT system.

Examples of resources are: web services, software modules with self contained procedures, persistent archives accessible via web service interfaces, computing and storage elements, compound service specifications, collections of objects.

In particular, a DILIGENT resource is anything whose related information must be gathered, stored, monitored, and disseminated in order to provide the valuable amount of knowledge needed during the creation and management of a DL as well as to operate the entire DILIGENT infrastructure.

When a DILIGENT resource owner registers a resource, he/she also provides a description of it. The services that implement the Resource Management functionalities presented in *D1.1.1 Test-bed functional specification report* complete this description by automatically gathering additional information. The full description is stored in an XML format and is maintained as information handled by the DIS-Registry service of the Collective Layer.

The goal of the DILIGENT Resources Model, presented in Figure 2, is to capture the structure and the main characteristics on the information maintained by the system about the available resources.

A *DILIGENTResource* is characterized by a unique identifier, allowing to unambiguously identify it, and a type, allowing to discriminate four main types of resources, i.e. DILIGENT Hosting Node (DHN), DL Component, Package, and gLite Service (see below for a detailed explanation of each resource type). Moreover, for each resource the system maintains the information related to the quality of the service supplied (*QoS)*. The semantic of this information as well the attributes captured are different in accordance with the different nature of resources.

QoS is expressed by a set of attributes reporting various quality aspects encountered in a distributed system [5]. Among the set of parameters that can be advertised by each DILIGENT resource there are:

- *Availability*, i.e. the probability that a resource can respond to requests;
- *Capacity*, i.e. the limit on the number of requests a resource is capable to handle;
- *Security*, i.e. the level and kind of security a resource provides;
- *Response time*, i.e. the delay from the request to getting a response;
- *Throughput*, i.e. the rate of successful request completion.

These attributes represent a preliminary set that will be enriched with further aspects specific for particular resources. Moreover, for each attribute the set of allowed values must be carefully identified; for instance an attribute may assume values in a discrete domain, while another one may assume values in a [0,1] continuous domain. Another aspect is related with the "quality" of the parameter advertised by the resource. The class *Measurement* is in charge to cover this aspect, in fact for each of the quality parameter measured it reports:

- The *certifiedBy*, that allows to express the "authority" responsible for expressing the attribute value; For instance, this can be an human as well as a service that is in charge to measure the parameter by monitoring the resource;
- The *valueImpact*, that allows to express the importance of this parameter in evaluating the quality of the resource;
- The *type*, that allows to discriminate among objective measurements automatically taken and subjective measurements expressed by humans.

The vocabulary of QoS parameters and the Measurement attributes needs further investigation and will be presented in detail in the D1.2.3 report due they depend on other DILIGENT services design.

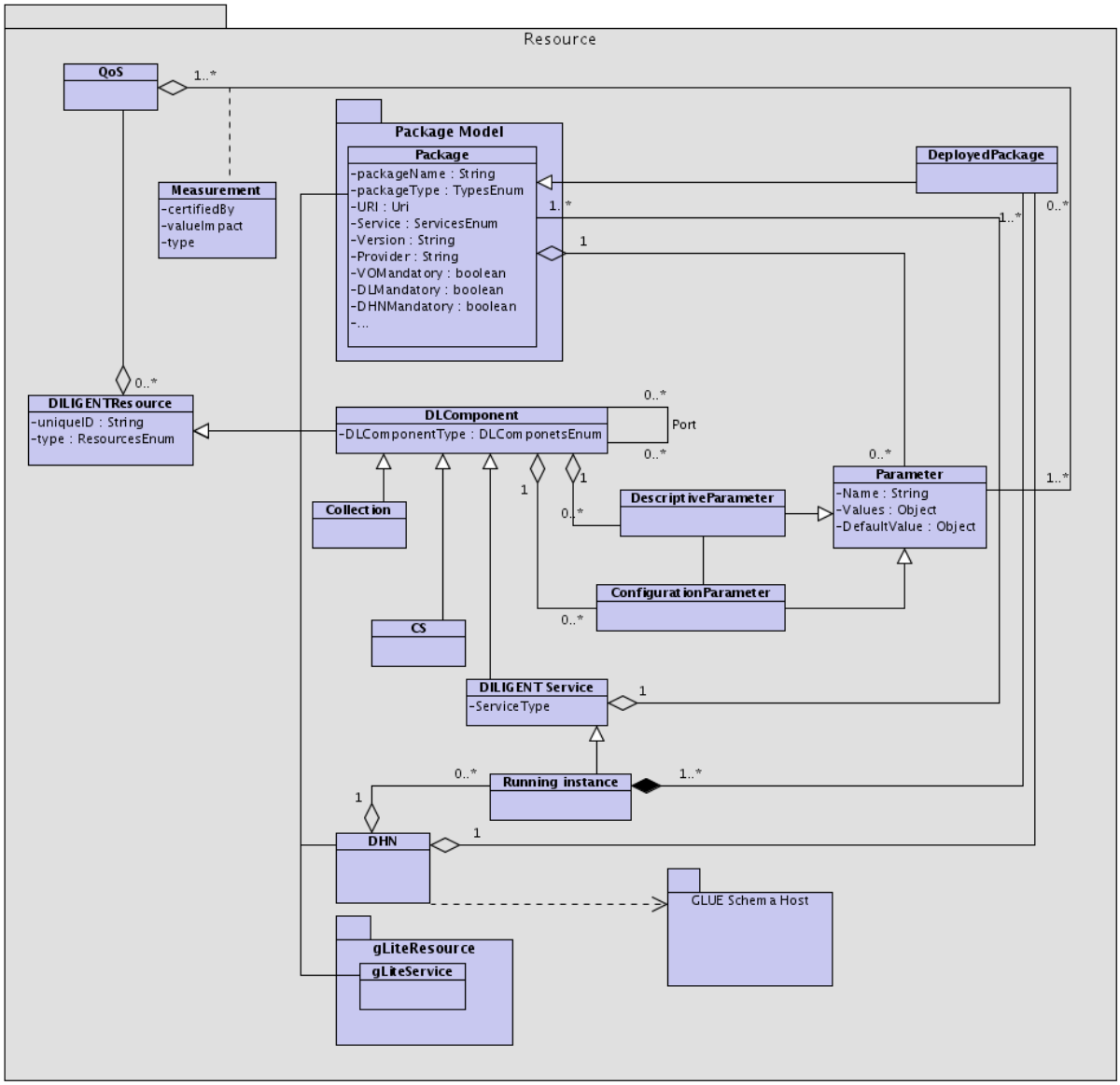

Figure 2. The DILIGENT Resources Model

The following resource types have been identified:

1. *Package*. In the DILIGENT context, a package is a "piece of software" that can be deployed in a DHN. In order to perform its tasks, the Keeper mainly uses all the information presented in Figure 2, plus other information presented by the Package Model in Section 3.1. The information maintained about packages are the name, the type, the URI where the package can be found, the service it belongs to, the version, the provider, and others that will be presented in detail in the D1.2.3 report. Moreover, for each package the system maintains i) its dependencies with respect to other packages, ii) the supported configuration parameters, and iii) the system requirements with respect to the node where the package is deployed. Within this context a WSRFService represents a particular type of package: it is the software that, when appropriately deployed, allows for the building of a DILIGENT Service.

2. *DLComponent*. This class of resources represents the components that can be used to build a DL. A DL Component aggregates a set of packages allowing to present them as a logical DILIGENT resource that can be used to define a DL.

   As described in detail in Section 7.5, the information that characterizes DLComponents are:

   - The type allows discriminating among the different components that can be used to build a DL. The high level types are *Collection*, *DILIGENT Service*, *Archive* and *Compound Service* (CS). However, a complete hierarchy allowing to classify the allowed type of DLComponents will be supplied in D1.2.3;
   - The list of attributes allowing to specify descriptive features of the component;
   - The list of attributes allowing to specify configurable features supported by the component;
   - The list of ports[2], i.e. relationships with other DLComponents that permit to express dependencies among components.

   The way the DLComponents will be modelled in terms of attributes and ports as well as the capability to express constraints on components composition need an in-depth investigation and will be presented in the D1.2.3 report.
3. *DHN*. It represents a hosting node, i.e. a computer connected to the network, which is available within the DILIGENT infrastructure and is capable to host DILIGENT Services. For the time being, the information maintained about this kind of resource is mainly intended for the matchmaking process and thus will be detailed in Section 5. However a good starting point is represented by the data reported in the GLUE Schema [4] about Host.
4. *gLiteService*. This type of resource models the information maintained about services built by instantiating gLite. A data model for these resources is reported in Section 3.2.

3.1 Package Model

In Figure 3 the Package Model is illustrated. A package is a "piece of software" that can be deployed on a DHN. The relevant information about a package can be divided in two types:

- *Descriptive information* that captures information that does not required a validation activity, like name, type, URI, version, provider.
- *Deployment information that needs to be validated by the DILIGENT system*. This second type of information can be further divided into two subtypes:
  - *Package dependencies*, enabling to express the need of a package to be deployed together with other packages in order to work properly.
  - *DHN requirements*, enabling to express requirements used by the Broker & Matchmaker to choice the DHN where the package should be deployed.

Moreover, for each package the parameter class allows to specify the pool of configuration parameters supported by the package. All types of packages belonging to DILIGENT inherit this general structure.

The DILIGENT packages have been grouped in four types as described below.

- *WSRFService*, representing a package that, once deployed, produces a Running Instance of a DILIGENT Service. For each WSRFService, the system captures data

[2] This name derives from the *component-port* approach adopted in configuration systems. More details are given in Section 7.5.

about the single file constituting it, the install/uninstall scripts and other information used to support the deployment process (see Section 6.5.2.3).

- *Portlet*, representing a package that once deployed produces a portlet that can be hosted by the DILIGENT Portal;
- *GridJob*, representing a package containing the code and the related information needed to run a certain job on the Grid (gLite);
- *Library*, representing a software library that can be hosted on DHN and is needed by other packages to support their tasks. There are at least two types of such libraries:
  - *Shared Library*, software library offering functionality of common utility, e.g. an XML parser library, a mathematical support library;
  - *Stub Library*, software library offering functionality for interacting with DILIGENT resources implemented as Web Services.

For a detailed description and an introduction on the usage of the model within the DILIGENT system see Section 6.5.

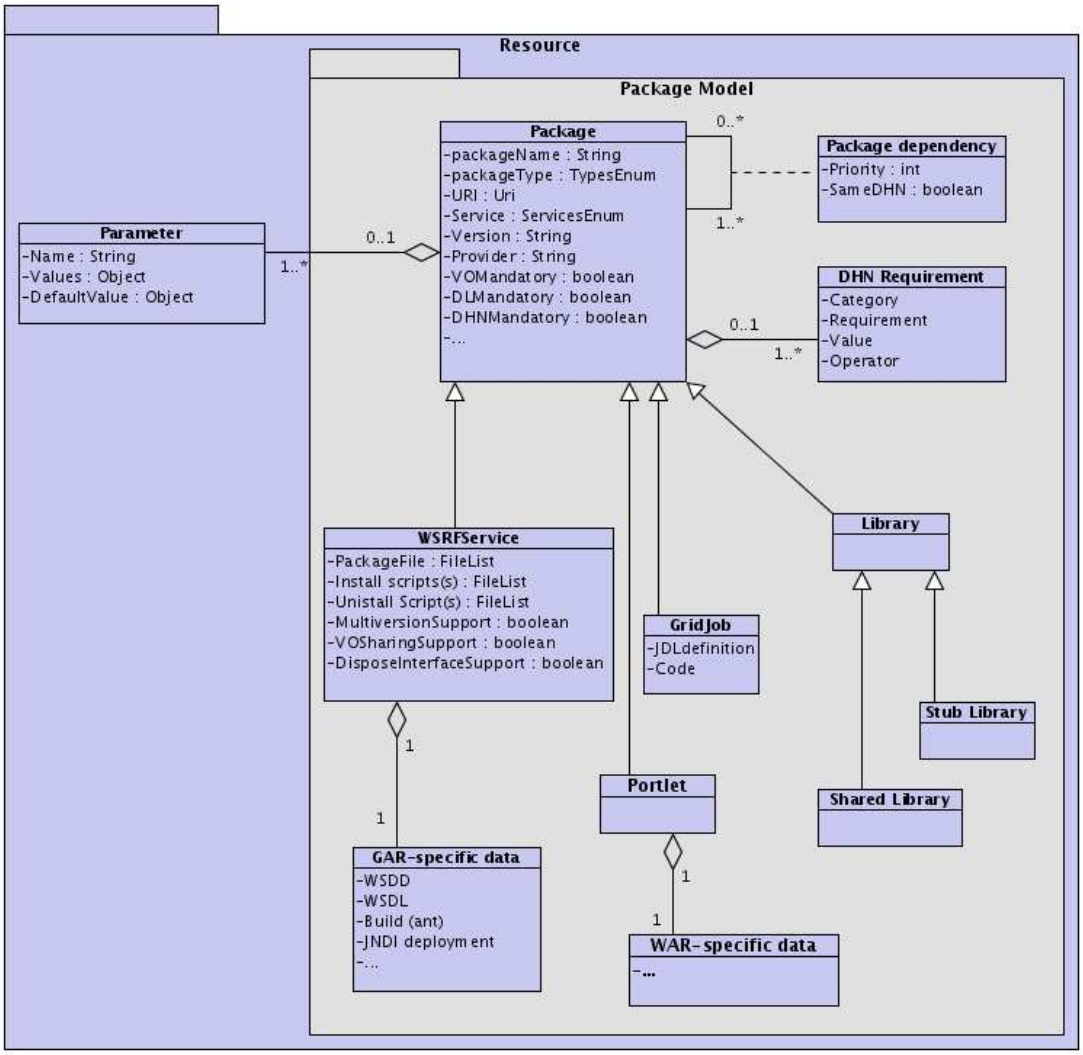

Figure 3. Package Model

3.2 gLite Resource Model

In Figure 4 [to be detailed in D1.2.3] the structure and the main characteristics on the information maintained by the system about the gLite resources are illustrated. For the time being and due to the ongoing design phase of the gLite middleware, this service

specification only reports the list of high-level services that are part of actual middleware release [7]:

- Authorization, Authentication and Delegation Services (as an integral part of the other subsystems);
- Computing Element (CE);
- File & Replica Catalog (called Single Catalog in this release – SC);
- File Transfer and Placement Service (Local Transfer Service);
- gLite I/O Server and Client;
- Logging and Bookkeeping Server (LB);
- R-GMA Servers, Client, Site Publisher, Service Tools and Service Discovery;
- Standard Worker node (WN, a set of clients and APIs required on a typical worker node installation);
- User Interface;
- VOMS and VOMS administration tools;
- Workload Manager System (WMS).

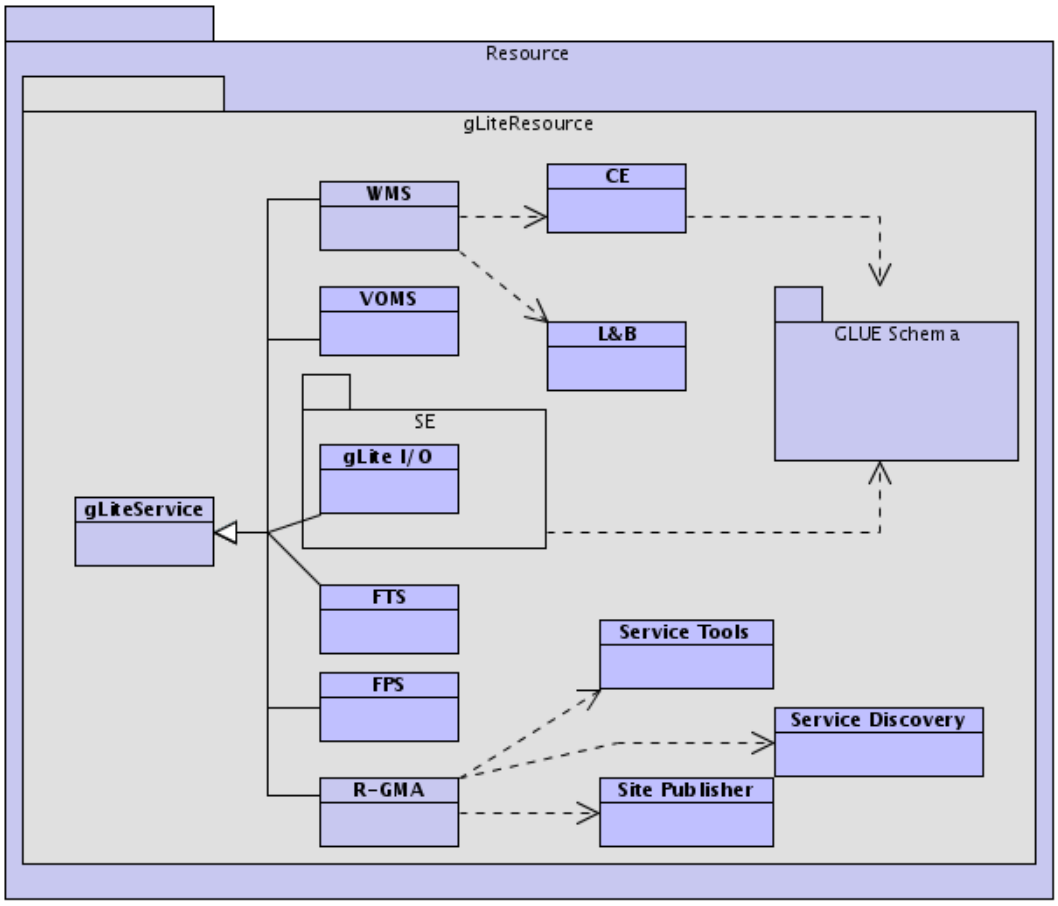

Figure 4. gLite Resource Model

The presented model needs to be revised and enriched with further details that will be acquired from newer gLite documentation. However, the information related with this type of resource is mainly obtained gathering the data that these resources advertise, and thus it strongly depends on their design.

Moreover, after having identified the information produced about these resources, the DILIGENT Information Service plans to enrich that pre-existing information with other attributes as well as appropriate mechanisms to do it; for instance, an appropriate procedure for QoS parameters estimation of gLite resources will be part of the detailed design description provided in the D1.2.3 report.

3.3 DILIGENT Security Model

Figure 5 shows the Security Model that DILIGENT Services and Clients can adopt to secure communication over the network. It is based on the security framework provided by Java WS Core. The aim of the model is to show security options available to DILIGENT Services and Clients though the security descriptor approach.

In the Java WS Core framework, each Service and Client can declaratively describe the security level to adopt during communication through an XML file named Security Descriptor. In DILIGENT each WSRF Service must be associated to at least one Service Security Descriptor. This file must contain information about authentication methods to use to securely contact the service. This information is used at deployment time by the Java WS Core container to correctly configure service security. This information is also registered in the DILIGENT Information Service in order to be used during package and service discovery.

Each method of the service can use different (and possibly multiple) authentication mechanisms (MessageSecurity, WSSecureConversation, Transport Layer Security (TLS)). Moreover, the file contains the list of Authorization handlers to use during enforcement of authorization policies. For a detailed description of the content of the Service Security Descriptor and some guidelines to use it in the DILIGENT architecture see Appendix A.

DHN and Clients (Stub Libraries) can also define their own Security Descriptors. For further information about how to create and use Security Descriptors, see the Java WS Core documentation[3].

[3] http://www.globus.org/toolkit/docs/4.0/security/authzframe/security_descriptor.html

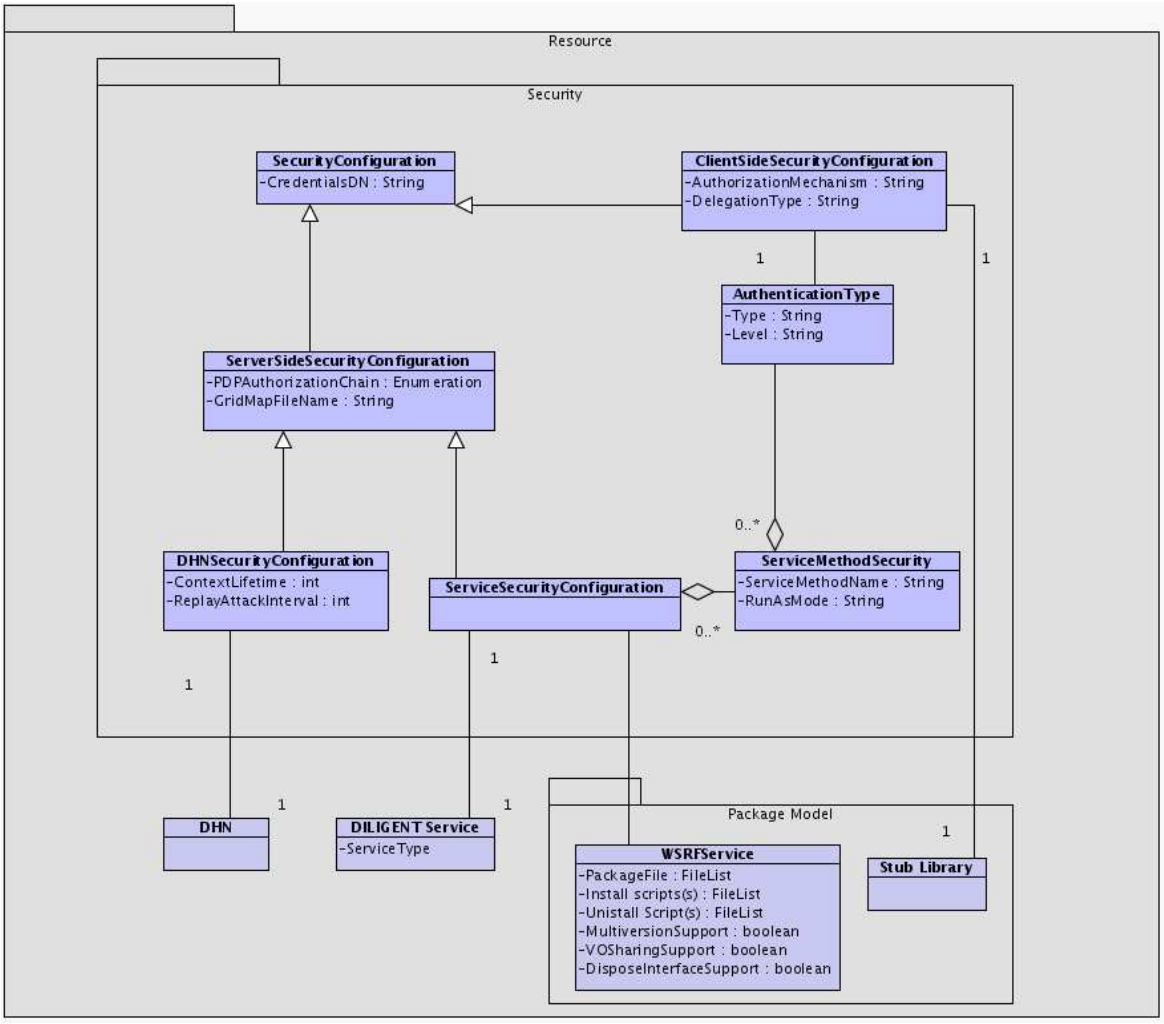

Figure 5. DILIGENT Security Model

4 INFORMATION SERVICE

4.1 Introduction

As in any distributed environment, the role of the information service in DILIGENT is to allow other services to be aware of the environment they operate in. The DILIGENT Information Service (DIS) maintains the most up to date information about the set of available distributed resources that compose the DILIGENT VO together with the status of the DILIGENT services.

In particular, the DIS provides mechanisms for:

- gathering, storing and supplying information about the resources needed by DILIGENT services, and
- monitoring the resources state information.

More in detail, it allows users and services to discover what resources are part of a Virtual Organization (VO) and to monitor those resources; it provides query and subscription interfaces to arbitrarily detailed resource information; it provides a trigger interface that can be configured to take action when pre-configured conditions are met; it archives information to allow historical query execution.

The DIS service acquires information through an extensible interface. This interface can be used (i) to query WSRF services for resource property information, and (ii) to execute a program capable to acquire information about the Grid infrastructure powered by the gLite middleware, mainly information on computational and storage resources.

It is designed to act as an information and monitoring service and therefore it is not an event handling system.

The DIS model is based on the Grid Monitoring Architecture (GMA) approach proposed by GGF[4] and on the Aggregator Framework software[5]:

- GMA is an abstract description of the components needed to build a scalable monitoring system; it models an information infrastructure as composed by a set of *producers* (that provide information), *consumers* (that request information) and *registers* (that mediate the communication between producers and consumers).
- Aggregator Framework is the software framework used to build services that collect and aggregate information. Based on it Index, Trigger, and Archive services can be built. The aggregator framework collects data from an *Aggregator Source* and sends that data to an *Aggregator Sink* for processing. All services use a common configuration mechanism to maintain information about which Aggregator Source to use and its associated parameters. Moreover they are self-cleaning, i.e. each registration has a lifetime: if a registration expires without being refreshed, the registration and its associated information are removed from the server. Aggregator sources distributed with the Java WS Core include modules that query service data, acquire data through subscription/notification, and execute programs to generate information. Currently available aggregator sinks include modules that implement WS Index service interface and WS Trigger service interface. Associated standards for the Aggregator Framework are the WS-ResourceProperties (WSRF-RP), WS-ResourceLifetime (WSRF-RL), WS-ServiceGroup (WSRF-SG), WS-Topics, and WS-BaseNotification.

[4] http://www.ggf.org/ and http://www-didc.lbl.gov/GGF-PERF/GMA-WG/
[5] http://www.globus.org/toolkit/docs/4.0/info/aggregator/

The rest of this section is organized as follows: Section 4.2 reports the DIS use-case view introducing the main functionalities provided by the service and how the DIS is organized in order to support them; Section 4.3 presents the logical view of the DIS architecture identifying the architecturally most significant components of the service; Section 4.4 reports a DIS deployment view by presenting a network configuration needed to deploy and run DIS components within the DILIGENT Infrastructure; Section 4.5 introduces design consideration and further details about the main services and components constituting the DILIGENT Information Service.

Some clarification about the DIS, the WS-Resource and the DILIGENT resource

In Section 3 we have identified the resources the DILIGENT system deals with. Moreover, the DILIGENT system is based on the WSRF framework. Within this framework a resource must be identifiable and must have zero or more properties expressible in XML (WS-ResourceProperties [22]). Such a resource is called a *WS-Resource* and "*is a Web service through which a resource can be accessed*". This means that to operate on a resource it must be bound to a service instance in order to form a WS-Resource. To be compliant with this framework we need to model the DILIGENT resources in terms of WS-Resources. In particular, from the DIS perspective, it is important to identify the Web Service that is in charge to provide the WS-ResourceProperties describing the DILIGENT resources:

- Package – The resource properties related to this kind of DILIGENT resource are provided by the Package Repository service (see Section 6.5.2.2);
- gLiteService – The resource properties related to this kind of DILIGENT resource are provided by the DIS-R-GMAClient service (4.5.2.5);
- DHN – The resource properties related to this kind of DILIGENT resource are provided by the Hosting Node Manager service (see Section 6.5.2.3);
- Collection – The resource properties related to this kind of DILIGENT resource are provided by Collection Service (the design of this service is part of D1.3.1 Content & Metadata Management services specification interim report);
- Compound Service – The resource properties related to this kind of DILIGENT resource are provided by Process Management service (the design of this service part of the D1.5.1 Process Management services specification interim report).

These resources need to have a lifetime greater than the WS-Resource that represents them. In particular their profile, i.e. the information provided at registration time compliant with the DILIGENT Resource model, must survive and be available in DILIGENT even in the case where there does not exist any deployed service in charge to actually manage the DILIGENT resource. As a consequence the DIS provides a service, the DIS-Registry described in Section 4.5.2.6, in charge to collect resources profile information, maintain them and then publish into the DIS via WS-ResourceProperties.

4.2 Use-Case View

The DILIGENT Information Service satisfies the following functionalities defined in D.1.1.1:

4.3 Resources Management

- 4.3.4 Edit Resource Profile
- 4.3.5 Store Resource Profile
- 4.3.6 Remove Resource Profile
- 4.3.7 Update Resource Profile

- 4.3.16 Search Available Resources
- 4.3.17 Get Available Resources
- 4.3.18 Browse Available Resources
- 4.3.19 Get Resource Status
- 4.3.20 Monitor a Resource

Moreover, the DIS supports the following functionalities from D.1.1.1 that are provided by other DILIGENT services:

4.2 DLs Management

- 4.2.2 Select Archives
- 4.2.3 Select Services
- 4.2.16 Analyze Available Resources
- 4.2.18 Create DL Resources
- 4.2.21 DL Resources Monitoring
- 4.2.25 Remove DL Resources

4.3 Resources Management

- 4.3.1 Add a Resource to DILIGENT
- 4.3.2 Register a Resource
- 4.3.8 Remove a Resource
- 4.3.9 Manage a Resource in a DL
- 4.3.10 Add a Resource to a DL
- 4.3.11 Create a DL Resource
- 4.3.13 Configure a Resource
- 4.3.14 Update a DL Resource
- 4.3.15 Remove a DL Resource
- 4.3.17 Get Available Resources

4.4 VOs Management

- 4.4.3 Add a Resource to a VO
- 4.4.18 Get User's VO Resources

In the use-case view presented in Figure 6 the DIS is composed by three main packages that respectively group functionalities of the:

- *resource information providers* (DIS-IP - stands for DIS-Information Provider),
- *resource information consumers* (DIS-HLS – stands for DIS-High-Level Service),
- *resource information collectors* (DIS-IC – stands for DIS-Information Collector).

Information providers publish/provide information about a DILIGENT resource. Each provider can adopt both the push and the pull model in order to provide its information. It

publishes information in the context of the Aggregator Framework using the WS-ResourceProperties standard[6].

Operating on top of a Grid infrastructure powered by the gLite[7] middleware, R-GMA, i.e. the gLite Information Service component, represents a particular kind of provider. This provider is in charge (i) to gather the information about Grid storage and computational resources constituting the Grid infrastructure, and (ii) to make available the gathered data to the DILIGENT services by publishing them on the DIS. Another particular kind of producer is represented by the ProvidePerNodeInfo: this producer is in charge to aggregate all the information provided by the single information providers hosted on the node and publish them on the DIS-IC.

Information consumers access the information via the high-level functionalities (DIS-HLS) provided by the DIS. The DIS-HLS package provides the following functionalities: discovery, notification, gets status and visualization. These functionalities support a customized level of authorization policies via "Enforce VO-level Authorization" UC provided by exploiting the "Check VO-level Authorization" provided by the Authorization Management area of the DVOS service (see Section 7.2).

The mediation between the producers and the consumers is done by the DIS-IC package. It plays a role similar to the Registry in the GMA model, even if it is in charge to supply the functionality to aggregate, manage, and provide resource information. In particular, taking into account that the aggregation functionality can be used to aggregate resource information provided by other aggregator sources, it is possible to organize multiple networks of aggregators based on replication and distribution mechanisms.

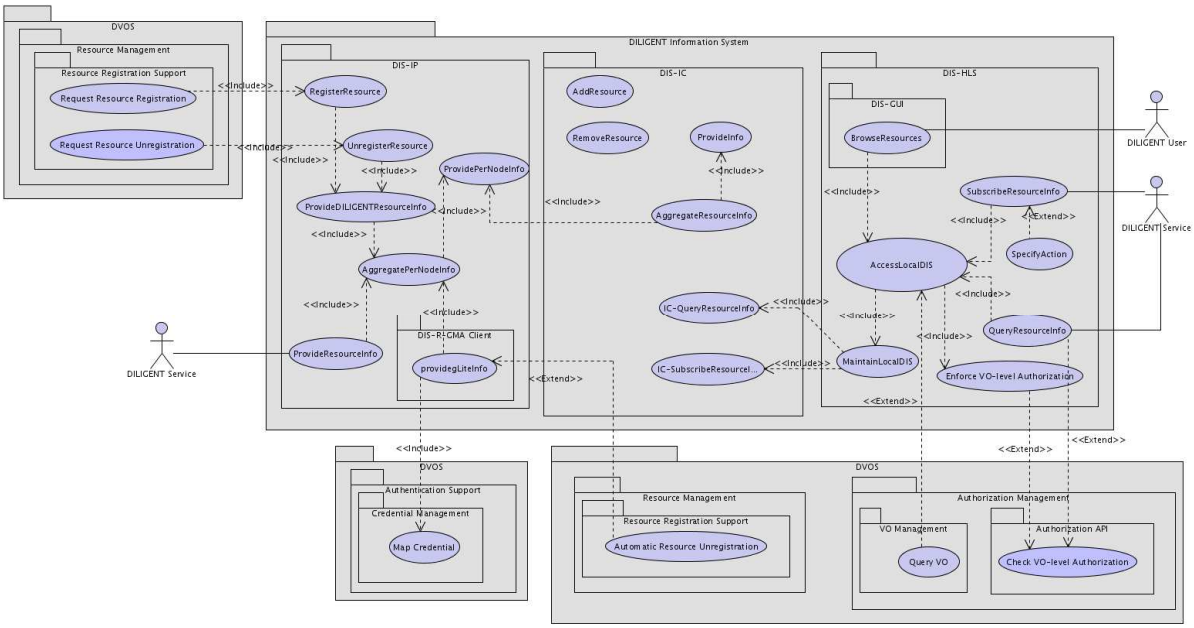

Figure 6. DIS - Use-Case view

Information Providers package (DIS-IP)

The Information Providers package includes all functionalities related to the publication/provision of resource information. Moreover, any time a new DILIGENT resource is created it must be registered in the DIS via the "RegisterResource" use case. At that time,

[6] http://www-128.ibm.com/developerworks/library/ws-resource/
[7] http://glite.web.cern.ch/

it must be specified the information production model adopted by the resource (push vs. pull). This package contains the following use cases:

- **RegisterResource** – This functionality is performed in order to register a DILIGENT resource.
- **UnregisterResource** – This functionality is performed in order to remove a DILIGENT resource from the DIS.
- **ProvideDiligentResourceInfo** – This functionality is performed in order to publish information about the DILIGENT resources described in section 3. More precisely, the information about Package, Collection, CS, and DHN resources are registered, unregistered, and published by the Resource Registration Support (see Section 7) that mediates between DILIGENT services and the DIS during the registration phase; DILIGENT Service and gLiteResource resources are registered, unregistered, and published by the Resource Registration Support by means of automatic functionality activated by the DL Management of the Keeper service and by the DIS-R-GMA client.
- **ProvideResourceInfo** – This functionality is performed by DLIGENT services in order to publish information on the DIS. This functionality covers both the push and pull modalities. In case of pull modalities the provider indicates the appropriate method that the DIS will invoke in order to acquire the information. In case of the push modalities it is up to the provider to notify the aggregator, as well as to provide the method that the DIS will invoke in order to actually acquire the information.
- **AggregatePerNodeInfo** – This functionality is performed on each hosting node of the DILIGENT infrastructure in order to aggregate the information provided by all the resources hosted on the node.
- **ProvidePerNodeInfo** – This functionality is performed on each node in order to publish (pull or push modalities can be used) the information about all the resources hosted on the node. The AggregatePerNodeInfo collects this information.

The DIS-R-GMA Client sub-package is responsible for collecting information about gLite resources. "ProvidegLiteInfo" use-case harvests information about the available storage and computational resources from the R-GMA, via the gLite R-GMA client, and then publishes/provides this data to the DIS.

Information Collector package (DIS-IC)

The Information Collector package includes all functionalities related to the storage, indexing, and managing of resource information. It contains the following UCs:

- **AddResource** – This functionality is performed in order to register a resource within an Information Collector Service.
- **RemoveResource** – This functionality is performed in order to remove a resource from an Information Collector Service.
- **AggregateResourceInfo** – This functionality is performed in order to collect information on resources provided by hosting node aggregators that in such a way mediate between information providers and DIS information collectors. Considering that a DIS information collector can also publish the information collected, through the ProvideInfo functionality, a DIS information collector, by performing the AggregateResourceInfo functionality, is able to gather data from other DIS information collectors, therefore allowing to build various level of aggregation.
- **ProvideInfo** – This functionality is performed in order to make available, with the same publishing mechanisms described for Information Providers based on WSRF specification, the resource information collected by the DIS information collectors. This functionality is needed in order to build hierarchies of information collectors.

- **IC-QueryResourceInfo** – This functionality is performed in order to enable services acting as clients to execute queries against the resource information. The query language supported is XPath. It is worth noting that if the DIS Information Collector is not capable to reply to a query by using the locally aggregated data and it is configured to use other DIS information collectors, the query will be forwarded to these instances following an intelligent peer-to-peer execution. Details on this activity are provided in further sections.
- **IC-SubscribeResourceInfo** – This functionality is performed in order to enable services acting as client to be notified whenever certain events are met. By providing this functionality the DIS-IC operates as the NotificationProducer while the subscriber will be a NotificationConsumer in accordance to the WS-Notification specifications.

High-level services package (DIS-HLS)

The High-level services package includes all functionalities related to the consuming of resource information. It is worth noting that these functionalities are provided by relying on those offered by the DIS-IC package. Many functionality are similar, the difference is the goal they are realized for, i.e. while the DIS-IC functionalities are designed to be used by the DIS-HLS, the DIS-HLS functionalities are designed to be use by actors external to the DIS, e.g. other services and human users. Note that executions of all use-cases of this package are subject to authorization from the DVOS service.

- **SubscribeResourceInfo** – A DILIGENT service can register itself to events notification by performing the "SubscribeResourceInfo" use-case, i.e. it subscribes to be notified when the information pertinent to a specific resource changes. The WS-Notification specifications that define a standard Web services approach to notification using a topic-based publish/subscribe pattern [29] have been investigated. They represent the foundation on which this functionality is based on.
- **SpecifyAction** - This functionality is performed when a SubscribeResourceInfo is executed in order to specify the action that must be executed when the subscription request for information is met.
- **QueryResourceInfo** – This functionality is performed in order to discover resources and gather their information. To discover resources the DILIGENT services perform queries (XPath) by specifying resource characteristics, i.e. any information made available by the DIS-IC package related to a resource.
- **AccessLocalDIS** – This functionality is performed in order to acquire information that is otherwise available by accessing an Information Collector. It is a sort of local cache maintained on each DHN in order to reduce response time. Notice that if the requested resource information is not cached locally, by performing the MaintainLocalDIS functionality, the information is first gathered and then used to reply to the request.
- **MaintainLocalDIS** – This functionality represents the actions performed in order to maintain the local cache up-to-date w.r.t. the information published globally on the Information Collectors. Notice that the local cache must contains the subset of the globally published information that is needed to satisfy the needs of locally-hosted services.
- **BrowseResources** – This functionality allows an authorized DILIGENT user to browse and view the resources and their related information. It is included in the package DIS GUI, i.e. the package representing the graphical user interface of the DIS.
- **Enforce VO-level Authorization** – This functionality is performed in order to have a customized level of authorization policies in accessing DIS to consume information.

It is provided by exploiting the "Check VO-level Authorization" provided by the Authorization Management area of the DVOS service (see Section 7.2).

4.3 Logical View

In this section we introduce the logical view based on the use-cases described. The logical decomposition of the DIS is shown in Figure 7. This logical view preserves the package names and decomposition presented in the use-case view. In the following a brief description of each package is provided.

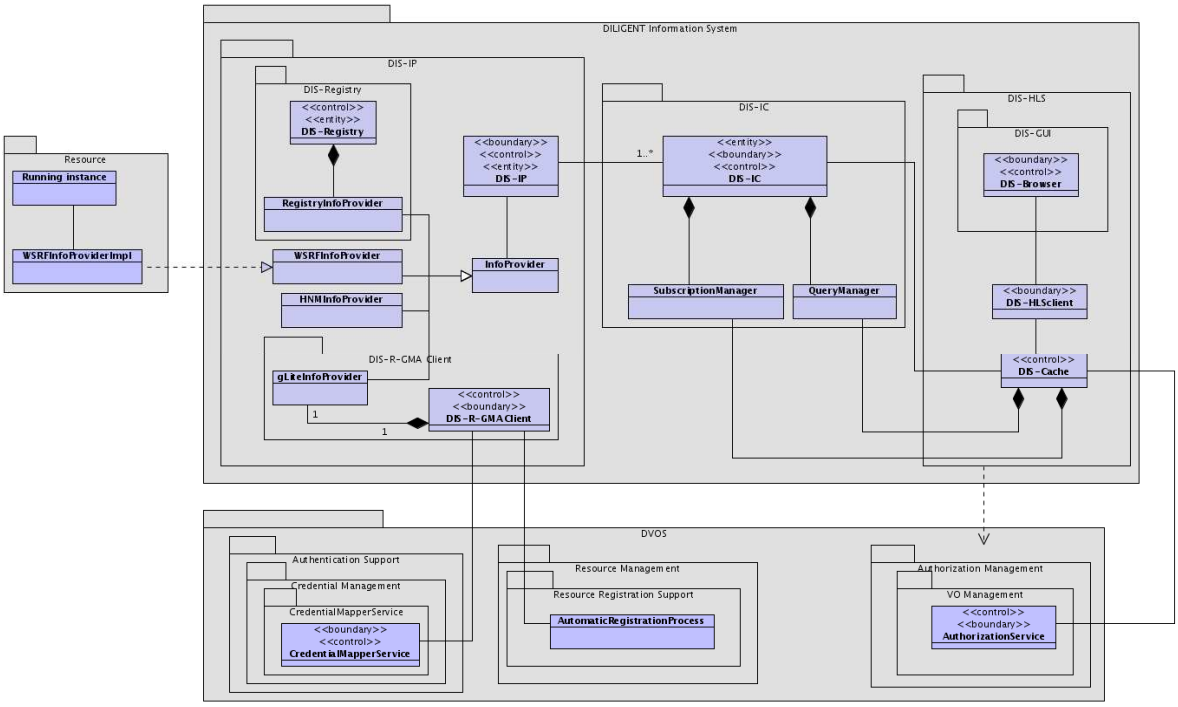

Figure 7. DIS - Logical View

DIS Information Provider package (DIS-IP)

This package contains classes and interfaces needed to publish the resource information in the DIS.

- **DIS-IP** – DIS-IP class is responsible for collecting status information from resource information providers located in the same DHN. Moreover it also provides the collected information to the DIS-IC that thus represents a second level of aggregation. For this reason and in accordance with the Aggregator Framework, it plays both the role of an Aggregator Source and of an Aggregator Sink. It is an aggregator sink w.r.t. the resources hosted on the node, while it is an Aggregator Source w.r.t. the DIS-IC.
- **InfoProvider** – This class is responsible for the creation, updating, and provisioning of resource properties. The realization of this class allows building software that acts both as Aggregator Source and as Information Provider. The former is a class that implements an interface (defined as part of the Aggregator Framework) to collect XML-formatted data. The latter represents the software used to effectively produce and update information. The DIS includes three Aggregator Sources:
  - (i) Query Aggregator Source, which polls a WSRF service for resource property information;
  - (ii) Subscription Aggregator Source, which collects information from a WSRF service via WSRF subscription/notification;

(iii) Execution Aggregator Source, which executes a supplied program to collect information.

Clearly any WSRF service that publishes resource properties is an InfoProvider. Other Info Provider specializations are the gLiteInfoProvider, the HNMInfoProvider, and the RegistryInfoProvider described below.

- **HNMInfoProvider** – This class represents a particular kind of Information Provider that is in charge to produce the information about the Hosting Node Manager (HNM) (see Section 6.5.2.3). In particular, at least the following information is supplied by this class:
  - Basic host data (name, ID);
  - Processor information;
  - Memory size;
  - OS name and version;
  - File system data;
  - Processor load data;
  - Software installed on the node; this information is quite heterogeneous, i.e. it can vary from software libraries to WSRF services that publish their properties. In the latter case, other useful information reported is start time, version, and service type name.
  - Other information needed by the Broker & Matchmaker to perform its matchmaking algorithm.
- **gLiteInfoProvider** – This class represents a particular kind of Information Provider in charge to produce the information about the resources that are monitored by the gLite R-GMA service. As previously stated, this functionality is needed in order to join an already existing Grid infrastructure based on gLite. This provider gathers information acquired via the DIS-R-GMAClient and provides them according to the schema for the computational and storage resources that we have decided to adopt in DILIGENT.
- **DIS-R-GMAClient** – The realization of this class allows to build a software module in charge to harvest information published by the R-GMA Server about computational and storage resources of the gLite based Grid infrastructure (CEs, SEs, I/O Servers, etc.).
- **RegistryInfoProvider -** This class represents a particular kind of Information Provider in charge to produce the information about the DILIGENT resources reported in Section 3. This provider gathers information from the DIS-Registry and provides them according to the schema for the DILIGENT resources that we have decided to adopt in DILIGENT.
- **DIS-Registry** - The realization of this class allows to build a software module in charge to provide registration and unregistration facilities for the DILIGENT resources (i.e. the DILIGENT resource profile described in Section 7) as well as their storage and preservation.

DIS Information Collector package (DIS-IC)

The classes belonging to this package provide the functionalities needed to collect resource information gathered through monitoring and discovery, and to publish this information in a common aggregator.

- **DIS-IC** – This class represents the main entity of the DIS-IC package. The realization of this class allows to build a software module in charge to collect, store,

and maintain resources information. Note that in the context of a running infrastructure, usually one or more DIS-ICs that collect data are deployed. Other facilities provided w.r.t. collected information are Subscription / Notification and Resource Property requests as described in the following classes.

- **SubscriptionManager** – This class represents the entity in charge to manage subscriptions. The realization of this class allows to build a software module to manage the Subscriptions about resource information. Subscriptions are based on WS-Notification specifications and other mechanisms provided by the Java WS core. Notice that this class is in charge to manage any kind of subscription received, i.e. subscription related to locally available information as well as subscriptions related to information stored by other DIS-ICs. In the latter case it is up to the SubscriptionManager to forward the request for notification to the appropriate DIS-ICs. To perform this task the class relies on configuration parameters the DIS-IC component has been instantiated with. These parameters contain also the URLs of the other DIS-IC instances the current DIS-IC instance can use to forward Subscription requests it is not capable to fulfill.
- **QueryManager** – This class represents the entity in charge to reply to queries on resource properties. The realization of this class allows to build a software module capable to match an XPath query against resource information. It is worth noting that this class is in charge to manage any kind of query received, i.e. queries related to locally available information as well as queries related to information stored by other DIS-ICs. In the latter case it is up to the QueryManager to forward the request for information to the appropriate DIS-ICs. To perform this task the class relies on configuration parameters the DIS-IC component has been instantiated with as well as on statistical data that can be collected at run time. These parameters contain also the URLs of the other DIS-IC instances the current DIS-IC instance can use to forward Query requests it is not capable to fulfill. Among the information collected at run time there are data related somehow to the content of each 'neighbor' DIS-IC. By relying on these data this class can improve the process of query forwarding.

DIS high-level services package (DIS-HLS)

Three main classes, DIS-HLSClient, DIS-Cache and DIS-Browser, are contained in this package. These classes model the different ways to access the resources information handled by the DIS-IC package.

- **DIS-HLSClient** – This class represents an entity enabling DIS clients to interact with the DIS in order to consume information. The realization of this class allows to build a software library to ease access to the information published on a DIS-IC. In particular, by interacting with this library, each Service is able to issue a query or a subscription to a DIS-IC, even without knowledge about its location. Actually, the class provides access to the DIS-Cache, i.e. an image of the information that is globally available on the DIS-ICs instances.
- **DIS-Cache** – This class represents an entity maintaining a local copy of the information globally published on the DIS, tailored to the information needs of the services hosted on the node. The realization of this class allows to build a web service that is in charge to form a local image of the information globally available on the DIS-ICs instances, and capable to satisfy the needs of the other services hosted on the node.
- **DIS-Browser** – This class represents the GUI of the DIS. The realization of this class provides a software module capable of displaying resource information via a standard web browser. The layout of the GUI can be easily modified and

personalized by manipulating the XSLT used during the rendering. Before implementing this class, an in depth investigation of the WebMDS component of the Globus toolkit must be performed.

4.4 Deployment View

In Figure 8 we present the deployment view of the components constituting the DIS. On the top left side of this diagram there is a node hosting a DILIGENT Service. The node contains the components of the DIS enabling the hosted service to produce information as well as to interact with the DIS in order to acquire the required information. Under this node there is another one where an instance of the DIS-Registry is deployed. This service acts as an information provider for DILIGENT Resource profiles. This information will be aggregated locally by the DIS-IP that is also in charge to provide it to the DIS-IC instances. On the right side of the picture there are two nodes: the upper one does not need to belong to the DILIGENT infrastructure; it only hosts the R-GMAServer. It represents the Information Service of a gLite-based Grid infrastructure DILIGENT joins with. The lower DHN hosts the DIS-R-GMAClient, i.e. the component in charge to gather information about Grid resources belonging the joined Grid infrastructure. This latter component, i.e. the DIS-R-GMA-Client, is a service and thus interacts with the DIS by relying on the same component, i.e. the DIS-IP, as the leftmost node previously described. The center node represent a DIS-IC instance. As previously stated, at instantiation time an arbitrary number of DIS-ICs can be deployed. All these instances are appropriately configured in order to appear to its users as a single service where all the published data can be retrieved and accessed. This means that the contacted DIS-IC may forward, client requests to other DIS-ICs, in a completely transparent way; it then collects and merges the results, and finally replies to its client with an answer capable to satisfy her/his needs.

In the following a brief description of the DIS components is provided.

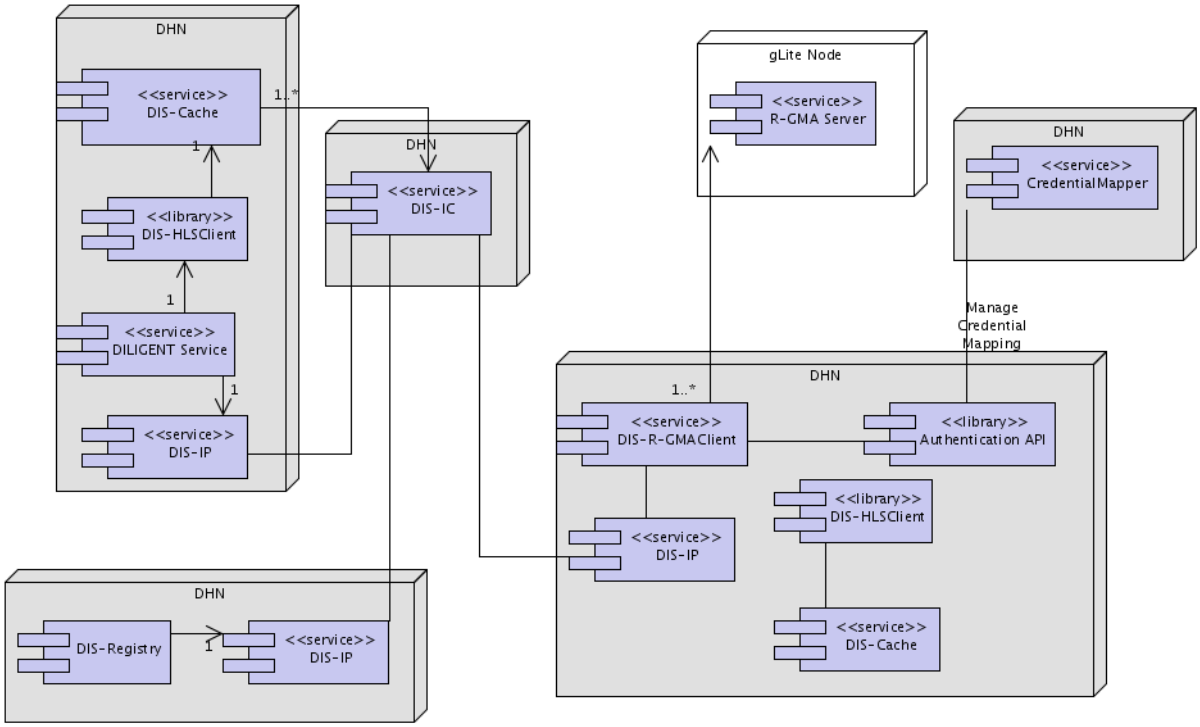

Figure 8. DIS - Deployment View

- **DIS-IP** – The DIS-IP is a web service that is deployed on each DHN. It implements the DIS-IP classes and thus it is responsible for gathering information about the resources stored on the node. As previously stated this service is based on the

Aggregator Framework software. As a consequence this framework must be available on each node. Thanks to the framework it gathers information about the DHN, the resources and services hosted on it, and any other information deemed useful; it then publishes this information, in the form of WS-Resource Properties, to the appropriate DIS-IC instance. Note that each DIS-IP is configured to have a priority list of DIS-IC to contact and will rearrange this list to overcome communication failures.

- **DIS-HLSClient** – The DIS-HLSClient is a library that is available on each DHN. It will be used by each of the services hosted on the node in order to have access to the DIS-IC. This library will enable locally hosted services to have consumer access to the information that is globally available within the DIS-IC instances hiding details about information partitioning among instances. Actually, this library is designed to operate on locally available information that is built and maintained by the DIS-Cache service.
- **DIS-Cache** – This service is in charge to build and maintain a local image of the information that is globally available on the various DIS-IC instances. The information that is locally available represents the subset of all the information that is needed to fulfil the information requirements of the services hosted on the node. Also this service is built by relying on the Aggregator Framework. Thus the DIS-Cache acts as an Aggregator Sink w.r.t. an Aggregator Framework where the Aggregator Sources are the DIS-ICs needed to fulfil the node needs. It is up to this service to rearrange and reconfigure the framework by adding new Aggregator Sources capable to fulfil information needs due to adjunction of other services on the node.
- **DIS-R-GMAClient** – This service represents the DIS-R-GMAClient and the gLiteInfoProvider classes reported in the Logical view. It is in charge to harvest resource information from the R-GMA Server it has been configured to interact with. This information is related to Grid resources (CEs, SEs, I/O Servers, etc.) belonging to the Grid infrastructure to join. Then, the gathered information is transformed in order to be compliant with the schema adopted in DILIGENT and then published on the DIS-IP where the DIS-R-GMAClient is deployed via the AggregatorFramework support.
- **DIS-IC** – This service represents the main aggregator of the information produced locally by nodes. It collects information produced by single resources via the DIS-IP service and makes them available via the WSRF resource property mechanisms.

  In order to avoid problems occurring in centralized solutions, like single points of failure, and to improve scalability, it is designed to operate in a distributed and replicated way. In particular, each DIS-IC instance can be configured to collect information from an established set of information sources (Aggregator Sources in Aggregator Framework terminology). Moreover, as each DIS-IC publishes collected information as resource properties it can also be considered as a kind of Aggregator Source and thus be used by another DIS-IC in the "usual" way. The DIS-IC provides also standard WSRF resource property query and subscription/notification interfaces to retrieve information. Note that the service contains also the logic to forward the query or the subscription to an appropriate DIS-IC instance if it is not able to reply to it using local information.
- **DIS-Browser** - This portlet represents the GUI of the DIS. It is a software module capable to display resource information via a standard web browser. The layout of the GUI can be easily modified and personalized by manipulating the XSLT used during the rendering. Thus different instances can have different layouts. When implementing it, an in depth investigation of the WebMDS software must be

performed. Another important aspect is related to the number of instances that can be created enabling each one to show a potentially different set of information in a different way.

4.5 Service design

4.5.1 Design considerations

The design of the DIS is based on the model proposed by the Grid Monitoring Architecture (GMA) presented by GGF. In particular, the identification of information producers (the DIS-IPs) and consumers (DIS-HLSs) as well as the presence of a registry component (DIS-ICs) is in line with this approach.

Another "constraint" that has driven the design of the DIS is related to the current implementations and software existing to implement Information Services in a Grid environment. In particular the Globus MDS4 and the gLite R-GMA have been studied. The former is taken into account because, to the best of our knowledge, the Globus toolkit is one of the most complete implementations of the WSRF specifications. Moreover, MDS4 provides a framework (Aggregator Framework) that can be used to implement *aggregator services*, i.e. services that collect information, maintain the collected information and provide mechanisms to access these information. DIS-IPs and DIS-ICs are classical aggregator services. The gLite R-GMA must be taken into account for a series of reasons. The main reason is related to the fact that DILIGENT represents an application running on a Grid infrastructure. This infrastructure is built by adopting the gLite middleware as enabling technology. This middleware, and thus the infrastructure, have an Information System based on R-GMA. The DILIGENT application, and in particular the DIS, must thus be able to interact with such kind of software. This last interaction is realized via the DIS-R-GMAClient.

An important issue to take into account in designing an Information Service is related to the application of a centralized or a distributed approach. Many Information Services are based on a centralized structure. This clearly introduces a number of problems, mainly scalability and tolerance to faults. Thus the most important point of the design that we want to highlight is related to the decision to conceive the DIS as a distributed and replicated component in order to avoid single points of failure and hotspots while accessing information. This design choice mainly reflects on the identification of different level of aggregation of information. We decided to have a per node aggregation performed via the DIS-IPs. Any other kind of aggregation can be realized by appropriately combining the DIS-ICs instances. An example of such an aggregation is reported in Section 4.5.3.

4.5.2 Components

In Section 4.4 we have presented the decomposition of the DILIGENT Information Service into six components. This architectural design is the most appropriate to obtain a DIS capable to adapt to and fulfil the needs of the DILIGENT system. In particular, it represents a solution whose goal is to obtain a good balance among factors like the distribution of the components and the performance of the system in order to overcome overhead due to network and protocol (e.g. SOAP) communications.

All these components are based on WS specifications, mainly WS-ResourceProperties, WS-BaseNotification, and WS-ServiceGroup. As a consequence many of the operation they provide are in line with these specifications.

Details on these components will be provided in the following subsections.

4.5.2.1 DIS-IP

The DIS-IP is a web service deployed on each DHN forming the DILIGENT infrastructure. It is responsible for collecting status information from resource information providers located in the DHN and for providing them to the DIS-IC instance the node is configured to communicate with.

4.5.2.1.1 State description

A DIS-IP manages information about the resources, i.e. Web Services and WS-Resources, hosted on the node. In particular, in accordance with the WS-ServiceGroup specification, it maintains an Entry for each of the resources hosted node. This entry is composed by two elements, the EndPoint Reference (EPR) of the resource and the content, an XML document containing information about the resource.

4.5.2.1.2 Operations

The most important high level operations offered by this service are:

- **add**(ProviderEPR) – This operation permits to add a new InformationProvider to a DIS-IP. Adding an InformationProvider means also to have a new WS-Resource that via its WS-ResourceProperties exposes the information provided by the provider.
- **getResourceProperty**(ResourcePropertyQNames) – This operation returns the value of each of the ResourceProperty specified in the list of property qualified names provided as parameter.
- **queryResourceProperty**(QueryExpr) – This operation allows a requestor to query resource properties using an XPath query expression.
- **subscribe**(ConsumerEPR, NotificationFilter) – This operation enables the DIS-IP to act as a NotificationProducer w.r.t. the DIS-ICs instances. A NotificationProducer produces a sequence of zero or more notifications. Via this operation the DIS-ICs can register to receive a subset of this sequence.
- **getCurrentMessage**(NotificationFilter) – This operation enables a consumer to retrieve the last notification published w.r.t. a given NotificationFilter. In WS-BaseNotification the notification Filter is based on another specification, the WS-Topics [23]. This specification defines a mechanism to organize and categorize the items of interest for a subscriber.
- **notify**(NotificationMessage) – This operation enables the DIS-IP to act as NotificationConsumer w.r.t. the services hosted on the node that act as NotificationProducers. This method is needed to allow Web services to provide information via the push modality. The NotificationMessage, in accordance to the WS_BaseNotification, contains (i) some metadata enabling the DIS-IP to be informed about the event that produce the notification like the EPR of the notification producer, the reference to the notification event and (ii) the message object of the notification.

4.5.2.1.3 Profile description

This section does not apply to this service.

4.5.2.1.4 Status Description

This service maintains a WS-Resource for each InformationProvider registered with it. As a consequence it exposes a WS-ResourceProperty for each InformationProvider. Three parts compose this property: the EPR of the DIS-IP, the EPR of the InformationProvider, and the resource properties provided by the InformationProvider

Moreover, as the DIS-IP acts as NotificationProducer, it exposes resource properties reporting the collection of topics supported (topics are used during the subscription phase).

4.5.2.1.5 Dependencies & Requirements

[to be provided in D1.2.3]

4.5.2.2 DIS-IC

This service represents the main aggregator of the information produced by remote DIS-IP instances about the resources locally hosted on each node. It represents a location where the data produced by single resources is collected and made available.

4.5.2.2.1 State description

The resources a DIS-IC manages are information about the resources provided by the DIS-IPs this service is configured to interact with. In particular, in accordance to the WS-ServiceGroup specification, it maintains an Entry for each of the resources provided by the DIS-IPs. This entry is composed by two elements, the EPR of the resource and the content, i.e. an XML document containing information about the resource.

4.5.2.2.2 Operations

The most important high level operations offered by this service are:

- **add**(ProviderEPR) – This operation enables to add an Information Provider to a DIS-IC. The Information Providers of a DIS-IC can be both a DIS-IP as well as a DIS-IC. Adding an Information Provider means also to have a new WS-Resource that via its WS-ResourceProperties exposes the information provided by the provider.
- **getResourceProperty**(ResourcePropertyQNames) – This operation returns the value of each of the ResourceProperty specified in the list of property qualified names provided as parameter.
- **queryResourceProperty**(QueryExpr) – This operation allows a requestor to query resource properties using an XPath query expression.
- **subscribe**(ConsumerEPR, NotificationFilter) – This operation enables the DIS-IC to act as a NotificationProducer w.r.t. other DIS-ICs instances or DIS-Cache instances. A NotificationProducer is capable to produce a sequence of zero or more notifications. Via this operation the DIS-ICs or the DIS-Caches can register to receive a subset of this sequence.
- **getCurrentMessage**(NotificationFilter) – This operation enables a consumer to retrieve the last notification published w.r.t. a given NotificationFilter. In WS-BaseNotification the notification Filter is based on another specification, the WS-Topics [23]. This specification defines a mechanism to organize and categorize the items of interest for a subscriber.
- **notify**(NotificationMessage) – This operation enables the DIS-IC to act as NotificationConsumer w.r.t. other DIS-ICs or DIS-IPs. This method is needed to provide information via the push modality. The NotificationMessage, in accordance to the WS_BaseNotification, contains (i) some metadata enabling the DIS-IC to be informed about the event that produce the notification like the EPR of the notification producer, the reference to the notification event and (ii) the message object of the notification.

4.5.2.2.3 Profile description

This section does not apply to this service.

4.5.2.2.4 Status description

This service maintains a WS-Resource for each resource registered on it by the DIS-IPs or by the DIS-ICs. As a consequence it exposes a WS-ResourceProperty for each of these resources. Three parts compose this property: the EPR of the provider (DIS-IP or DIS-IC), the EPR of the resource InformationProvider, and the resource properties provided by the InformationProvider

Moreover, as the DIS-IC acts as NotificationProducer, it exposes resource properties reporting the collection of topics supported (topics are used during the subscription phase).

4.5.2.2.5 Dependencies and Requirements

[to be detailed in D1.2.3]

4.5.2.3 DIS-HLSClient

4.5.2.3.1 Description

The DIS-HLS Client is a library that is distributed on each of the DHN. Its goal is to provide facilities to the DILIGENT resources hosted on the DHN to access the information stored on the DIS-ICs. It is worth nothing that the operations performed locally on the node are actually executed on a global scope, i.e. the library provides access to the information globally available on the DIS.

The most important high level operations performed by this library are:

- **getResourceIDs**(QueryExpression) – This operation allows to retrieve all the available resources registered on the DIS whose properties match against the QueryExpression. This QueryExpression is an XPath expression.
- **listAvailableResource**(ResourceType) – This operation allows to retrieve all the available DILIGENT resources of a specified type, e.g. all the Collection, all the Archive, etc.
- **getResourceInfo**(ResourceID) – This operation allows to retrieve the published information about a resource.
- **subscribe**(ConsumerEPR, NotificationFilter) – This operation enables to register the client as NotificationConsumer w.r.t. information published on the DIS.

[to be detailed in D1.2.3]

4.5.2.3.2 Usage

This library is designed to be used by any service that needs to act as consumer w.r.t. the DIS. Actually, the library provides access to the information stored locally on each DHN within the DIS-Cache. The library eases the access to these information by providing some high level operations.

[to be detailed in D1.2.3]

4.5.2.4 DIS-Cache

This service is in charge to build and maintains a local image of the information that is globally available on the various DIS-IC instances. The information that is locally available represents the subset of all the data that is needed to fulfil the information requirements of the services hosted on the node.

4.5.2.4.1 State description

The DIS-Cache manages information about the resources published on the DIS-ICs instances it is configured to interact with and provided by these instances. However, we decided to not manage this information as a WS-Resource and thus expose it via WS-

ResourceProperties because the designed interaction in accessing this information is not Service-2-Service. We have designed a library to be hosted locally and thus we do not need the facilities offered by the WSRF specifications.

4.5.2.4.2 Operations

The most important high level operations performed by this service are:

- **getResourceProperty**(ResourcePropertyQNames) – This operation returns the value of each of the ResourceProperty specified in the list of property qualified names provided as parameter. Worth noting that if this information is not stored locally, the service forwards the request to the appropriate DIS-IC instance, stores the retrieved information locally and then replies to the request.
- **queryResourceProperty**(QueryExpr) – This operation allows a requestor to query the resource properties using an XPath query expression. Worth noting that if this information is not stored locally, the service forwards the request to the appropriate DIS-IC instance, stores the retrieved information locally and then replies to the request.
- **subscribe**(ConsumerEPR, NotificationFilter) – This operation enables a client to specify a subscription request on information published on the DIS. Worth noting that these requests are forwarded to the appropriate DIS-IC.
- **getCurrentMessage**(NotificationFilter) – This operation enables a consumer to retrieve the last notification published w.r.t. a given NotificationFilter. In WS-BaseNotification the notification Filter is based on another specification, the WS-Topics [23]. This specification defines a mechanism to organize and categorize the items of interest for a subscriber.

4.5.2.4.3 Profile description

This section does not apply to this service.

4.5.2.4.4 Status description

As previously stated this section does not apply to this service.

4.5.2.4.5 Dependencies and Requirements

[to be provided in D1.2.3]

4.5.2.5 DIS-R-GMAClient

This service represents the DIS-R-GMAClient and the gLiteInfoProvider classes reported in the Logical view. It is in charge to harvest resource information from the R-GMA Server it has been configured to interact with. This information is related to Grid resources (CEs, SEs, I/O Servers, etc.) belonging to the Grid infrastructure to join. Then, the gathered information is manipulated in order to be compliant with the schema adopted in DILIGENT and then published on the DIS-IP where the DIS-R-GMAClient is deployed via the AggregatorFramework support.

4.5.2.5.1 State description

This service harvests and maintains information about Grid resources registered on an R-GMA Server. As this service acts as an Information Provider it maintains a resource for each of these resources and exposes the appropriate information about them.

4.5.2.5.2 Operations

The most important high level operations performed by this service are:

- **getResourceProperty**(ResourcePropertyQNames) – This operation returns the value of each of the ResourceProperty specified in the list of property qualified names provided as parameter.
- **subscribe**(ConsumerEPR, NotificationFilter) – This operation enables the DIS-R-GMAClient to act as a NotificationProducer w.r.t. the DIS-IP instance. As NotificationProducer it is capable to produce a sequence of zero or more notifications. Via this operation the DIS-IP can register to receive a subset of this sequence.
- **getCurrentMessage**(NotificationFilter) – This operation enables the DIS-IP to retrieve the last notification published w.r.t. a given NotificationFilter. In WS-BaseNotification the notification Filter is based on another specification, the WS-Topics [23]. This specification defines a mechanism to organize and categorize the items of interest for a subscriber.

4.5.2.5.3 Profile description

This section does not apply to this service.

4.5.2.5.4 Status description

This service maintains a WS-Resource for each Grid resource registered on the R-GMA Server it is configured to interact with. As a consequence it exposes a WS-ResourceProperty for each of these resources. Two parts compose this property: the EPR of the Grid resource, and the resource properties harvested by this service and provided by it.

Moreover, as the DIS-IC acts as NotificationProducer, it exposes resource properties reporting the collection of topics supported (topics are used in subscription phase).

4.5.2.5.5 Dependencies and Requirements

[to be provided in D1.2.3]

4.5.2.6 DIS-Registry

This service provides registration and unregistration facilities for the DILIGENT resources (i.e. the DILIGENT resource profile described in Section 7) as well as their storage and preservation.

4.5.2.6.1 State description

This service maintains the resource profile for each DILIGENT resource available in DILIGENT.

4.5.2.6.2 Operations

The most important high level operations performed by this service are:

- **add**(ResourceProfile) – This operation allows to add a new resource profile representing a new available DILIGENT resource. Adding such new resource actually means also to create a new WS-Resource (acting as placeholder of the resource registered) that exposes the information via its WS-ResourceProperties.
- **update**(ResourceID, ResourceProfile) – This operation allows to update the information publicly available on the DIS about a DILIGENT resource.
- **remove**(ResourceID) – This operation allows to remove the resource from those publicly available on the DIS.
- **getResourceProperty**(ResourcePropertyQNames) – This operation returns the value of each of the ResourceProperty specified into the list of property qualified names provided as parameter.

- **subscribe**(ConsumerEPR, NotificationFilter) – This operation enables the DIS-Registry to act as a NotificationProducer w.r.t. the DIS-IP instance and any other service. A NotificationProducer is capable to produce a sequence of zero or more notifications. Via this operation any service, and in particular the DIS-IP, can register to receive a subset of this sequence.
- **getCurrentMessage**(NotificationFilter) – This operation enables a consumer to retrieve the last notification published w.r.t. a given NotificationFilter. In WS-BaseNotification the notification Filter is based on another specification, the WS-Topics [23]. This specification defines a mechanism to organize and categorize the items of interest for a subscriber.

4.5.2.6.3 Profile description

This section does not apply to this service.

4.5.2.6.4 Status description

The service exposes a WS-ResourceProperty that is composed by the list of all the profiles of the DILIGENT resources available within DILIGENT. Each item of this list is composed by at least two elements: (i) the EPR of the resource representing the DILIGENT resource and (ii) the content, i.e. an XML document reporting the information about the DILIGENT resource deemed as relevant within DILIGENT.

4.5.2.6.5 Dependencies and Requirements

[to be provided in D1.2.3]

4.5.3 Deployment scenario(s)

Figure 9 depicts a possible deployment scenario of the components forming the DIS. This scenario is based on the distribution of the information among the various DIS-ICs based on the concepts of DL and VO. In particular, in this scenario we decided to have three levels of aggregation: per node, per DL, per VO. Per node aggregation is performed via the DIS-IPs services instantiated on each of the DHN. Per DL aggregation is performed by instantiating a DIS-IC service for each DL. This service is in charge to aggregate data coming from the DHN hosting the services forming the DL. The third level of aggregation, per VO aggregation, is performed by instantiating a DIS-IC service instance configured to gather data coming from each DL DIS-IC. Finally, notice that a generic client node (the top-right node) is entitled to have access to the information aggregated at different levels.

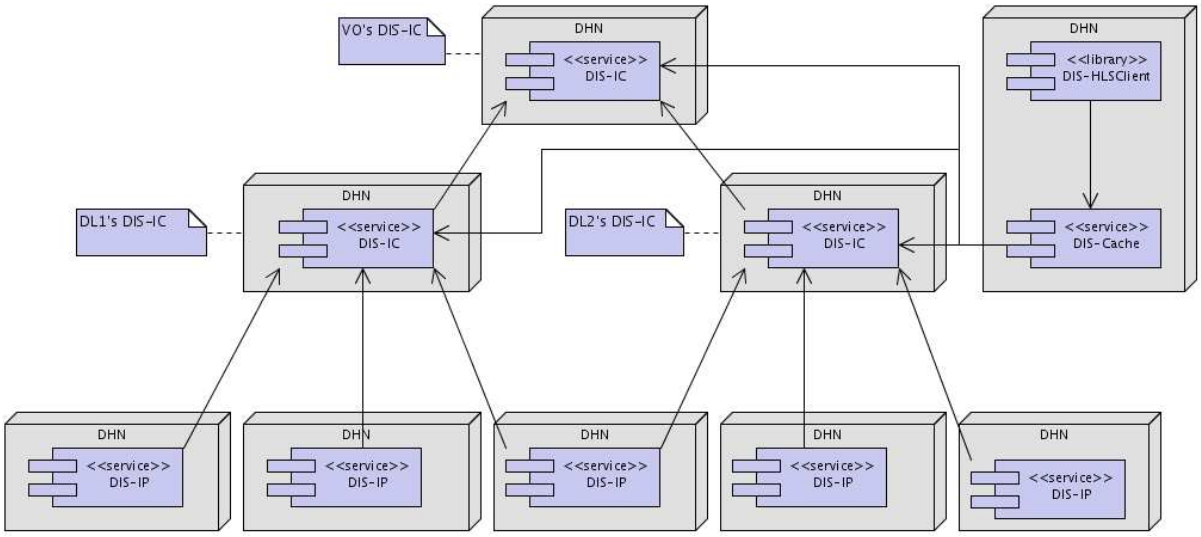

Figure 9. DIS - An aggregation scenario

5 BROKER & MATCHMAKER SERVICE

5.1 Introduction

The Broker & MatchMaker Service (BMM) is in charge of supporting the Keeper service in deploying a new Digital Library on a set of DILIGENT Hosting Nodes (DHNs). In particular, once the Keeper has identified the set of packages needed to build a new DL (more generally, any time it needs to deploy a new package, see Section 6.1.1), their requirements and their relationships, the BMM's job is to identify a set DHNs to be used as target hosts for the deployment.

5.2 Use-Case View

Deliverable D1.1.1 (Functional Specification) identifies the 'findOptimalAllocation' as the high level functionality mapped on the Broker & MatchMaker (ref. D1.1.1 "Test-bed Functional Specification" Section 4.3.2).

The main consumer of this functionality is the Keeper service that, when deploying a new Digital Library, needs to know where to deploy the set of packages required for a new Digital Library or Virtual Organization. To return such a configuration to the Keeper, the Broker & MatchMaker performs a matching between package requirements (for each package to be deployed) and the capabilities of DHNs currently available to the VO.

The Broker & MatchMaker doesn't gather the status of the whole DHNs environment by itself; rather, for its computations, it relies on information supplied by the DILIGENT Information System (DIS). The status of DHNs, periodically fetched from the DIS or notified by it upon any change in DHNs, is stored in the Broker & MatchMaker Catalogue.

The following diagram presents the main functionalities of the service:

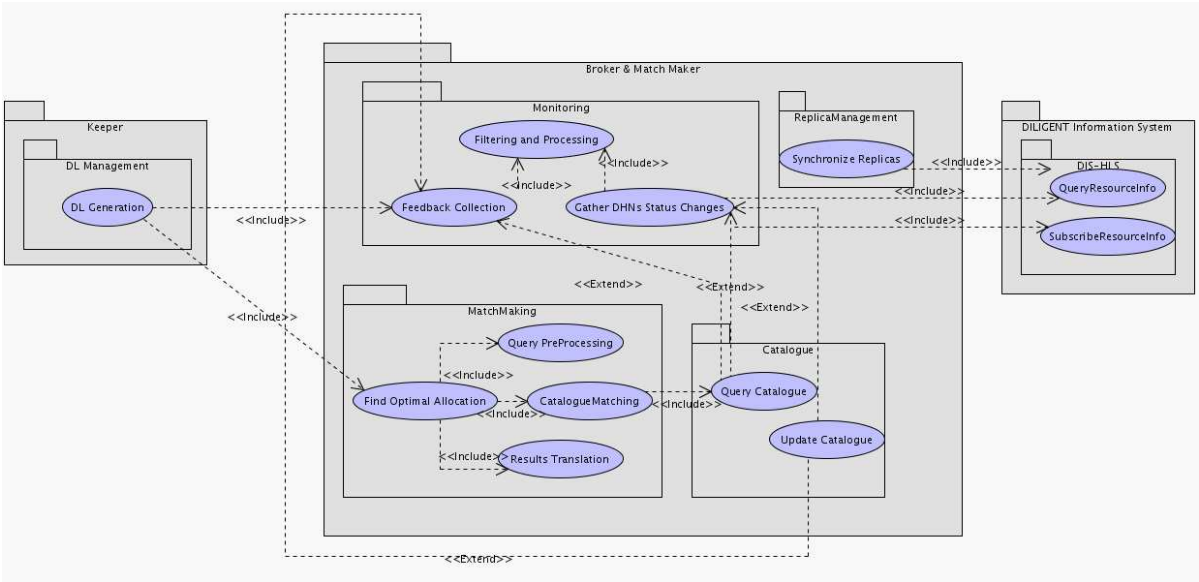

Figure 10. Broker & MatchMaker - Use-Case View

MatchMaking Package

This package groups functionalities related to the identification of a possible deployment solution after a request from the Keeper.

- **Find Optimal Allocation** – This is the main functionality of the BMM; it enables DILIGENT services (typically within the DILIGENT Collective Layer, in particular the Keeper) to discover candidates DHNs in order to deploy packages conforming to the

Package Data Model described in 3.1. This functionality is based on three further internal functionalities described hereafter.

- **Parse Request** – This functionality deals with the parsing of requests and the creation of an internal representation suitable to be submitted to the matching algorithm.
- **Match** - This is the core functionality of the Broker & Matchmaker Service. It's in charge of finding and evaluating alternative deployment configurations on available DHNs.
- **Present Results** – Generates a representation of the deployment solution identified by the matching algorithm.

Monitoring Package

The Monitoring Package groups functionalities related to the management of an up-to-date snapshot of the physical DILIGENT infrastructure; it also has to keep track of successful/faulty past deployments in order to improve the quality of a subsequent deployment solutions

- **Gather DHN Status Changes** - the BMM subscribes itself to the DIS in order to be notified about any DHNs status changes. This functionality is realized through the subscription/notification mechanism provided by the DIS.
- **Processing and Filtering -** Notifications are internally processed and filtered. The Processing phase parses the notification content and generates an internal representation of the DHN status suitable to be submitted to the Catalogue for storage; at the same time, Filtering is applied to notifications to remove any information that is not useful for matching purposes. Finally, this functionality also deals with the generation of derivative information about DHNs not included in notifications from the DIS (e.g. last-day average CPU load).
- **Feedback Collection** - Even if a configuration matches all the requirements made by a requester, a deployment action based on this configuration, may lead to a failure. This can happen for several reasons, i.e. status changes on the DHN between the moment the BMM identifies a configuration and the actual deployment action, making some resources actually unavailable on the node; the package cannot be deployed on the node because of software conflicts not modelled within DILIGENT, etc.; according to the failure notification received, the BMM can perform different actions: force refresh of DHNs status, rank down the reliability of a DHN, etc.

Catalogue Package

- **Update Catalogue** - This use case models the internal storage of feedbacks from the Keeper and notifications from DIS after the *Processing and Filtering* stage.
- **Query Catalogue** – This functionality models the retrieval of DHNs status and any information previously stored useful for the matching activity. This activity is performed any time a new deployment request has to be processed.

ReplicaManagement Package

- **SynchronizeReplicas** – This use case models the synchronization of BMM replicas within a VO with respect to feedbacks received from the Keeper

5.3 Logical View

Broker & MatchMaker architecture is composed of five main packages:

- a *catalogue* package maintaining an up-to-date picture of the DHNs status,
- a *matchmaker* package which is responsible for matching the service request with the catalogue content and providing the requester with a possible deployment solution,
- a *monitoring* package which is in charge of keeping the BMM catalogue up-to-date with respect to notifications received from the DIS and feedbacks from the Keeper,
- a *replicaManagement* package ensuring the whole set of BMM replicas in the VO shares the same model,
- a *BMM-API* package allowing clients to use functionalities provided by this service.

No Graphical User Interface is provided by the Broker & MatchMaker as this service is not meant to interact with human users.

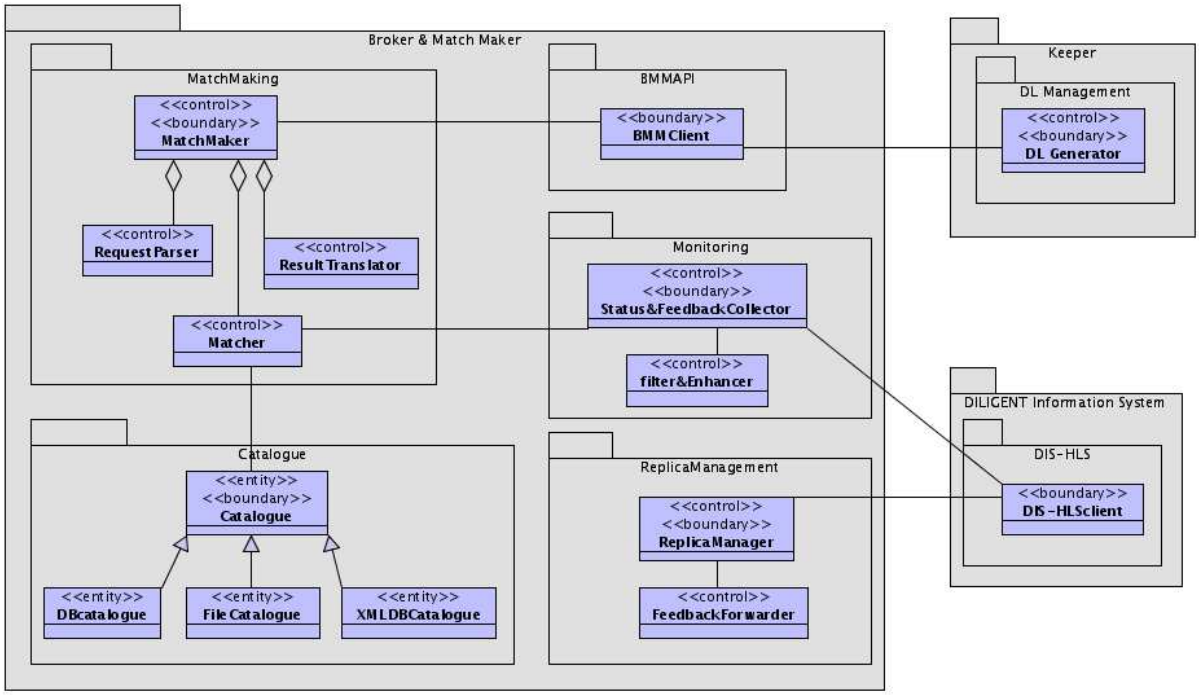

Figure 11. Broker & Match Maker - Logical View

Catalogue package

The Catalogue package is charge of maintaining the most up-to-date status of DILIGENT Hosting Nodes, as well as information derived from feedback received from the Keeper.

- **Catalogue** – Any class implementing this interface should be capable of maintaining and allowing the retrieval of status information for each DHN in the VO. The catalogue also manages additional information useful for matching purposes (e.g. failure feedbacks from the Keeper). Three alternative implementations of the Catalogue interface are provided by the Broker & MatchMaker: DBCatalogue, FileCatalogue and XMLDBCatalogue.
- **DBCatalogue** – This implementation of the Catalogue interface exploits a relational database as persistence support for DHNs status. This is the choice of preference for big VOs (i.e. a large number of DHNs) because of the better persistence support and high performance of retrieval of information. On the other hand, this approach requires the availability or the deployment on-demand of a DBMS on DHN accessible from the the DHN where the BMM is being deployed.
- **FileCatalogue** – This implementation of the Catalogue interface stores DHN's status in the local file system. Due to its simplicity and independence of other technologies,

this implementation is most suitable for small VOs (i.e. few available DHNs) enablig easier dynamic deployment of the Broker & Match Maker component.

- **XMLDBCatalogue** – This implementation of the Catalogue interface exploits a native XML database as persistence support for DHNs status. This solution provides a natural way of maintain XML-based DHN's status descriptions.

ReplicaManagement package

This package is concerned with ensuring that each BMM replica in a VO behaves as the others in terms of the solutions proposed. The issue in ensuring synchronization is actually in circulating feedbacks from the Keeper because of all the replicas adopt the same matching algorithms and obtain the status of each DHN from the DIS in the same way.

- **ReplicaManager** – This class keeps track of BMM instances available for the VO. References to BMM replicas are identified by using the discovery functionality of the DIS
- **FeedbackForwarder** – This class in charge of forwarding feedbacks received from to the whole federation of BMMs for the VO.

BMM-API package

- **BMMClient** – This class provides access to BMM functionalities, avoiding the need to select and contact a BMM Service replica. For this, the BMMClient interacts with the DIS-HLS client class in order to discover the location of a BMM.

MatchMaking package

This package models the core functionalities of the BMM. It actually interacts with external services by accepting requests and responding with a possible configuration. It is composed by four main classes:

- **MatchMaker** is the entry point for the MatchMaking package and manages the whole matching process;
- **RequestParser** realizes the Parse Request functionality creating an internal representation for the incoming request;
- **ResultTranslator** is in charge of translating results of the matchmaking process to an XML representation to be returned to the requester. When no matching configuration is found, the MatchMaker may optionally reply with some details about the reason for that (e.g. no DHN equipped with asked requirements);
- **Matcher** is the main class of the package. It implements a matching algorithm and is responsible for determining a possible deployment solution. Details about the problem being addressed and the solutions adopted are provided in the Service Design section.

Monitoring Package

DIS notifications about DHN status changes and Keeper feedbacks are collected here. After an adequate filtering and processing phase, these notifications can trigger catalogue updates. Finally, Monitoring deals with the gathering of deployment failure feedbacks received by the Keeper, required for ranking tuning.

- **Status&FeedbackCollector** is in charge of:
    - discovering VO's DHNs and retrieving DHNs status at start-up time;
    - exploiting the DIS subscribe/notification mechanisms to be notified about any change in DHNs status and DHN topology;
    - collecting direct failure feedbacks from the Keeper and forwarding feedbacks from BMMs replicas in the VO;
    - accessing the local BMM catalogue to persist status and feedbacks.

- **Filter&Enhancer** provides the Status&FeedbackCollector class with the procedures needed to (i) filter of unneeded received notifications and (ii) create the aggregate and derivative information (e.g. total available disk space, last-day average cpu load, etc.)

5.4 Deployment View

From a deployment point of view, two components have been identified. As depicted in Figure 12 these components are designed to be hosted in different networked locations:

- MatchMaker service – this is a VO-specific component. To improve availability, multiple instances of this service can be deployed on different DHNs. The number of instances and persistence support type is determined at VO creation time by the Keeper according to performance and reliability needs. As explained in the Logical View section, each instance updates its own copy of the DHNs environment status after DIS notifications, thus ensuring consistency of the model. Failure feedbacks, conversely, are circulated using replicas within the federation, adopting a push and pull model.
- BMM-API – provides local access to BMM functionalities, avoiding the need to select and contact a BMM replica. This library is typically deployed together with the DL Management Service (see Section 6.5.2.1).

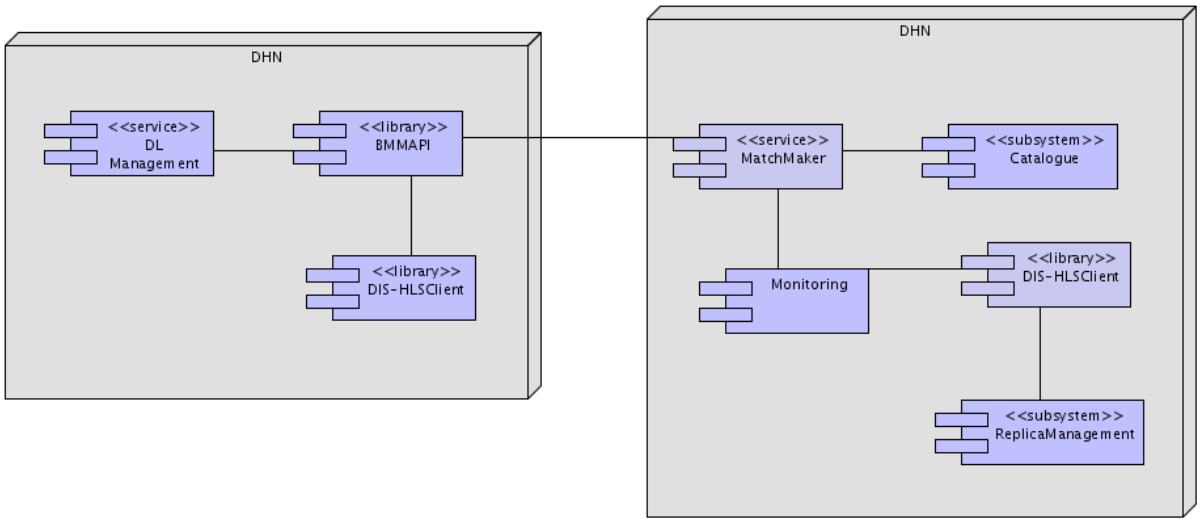

Figure 12. Broker & Match Maker - Deployment View

5.5 Service design

5.5.1 Design considerations

5.5.1.1 The Problem of Matching

The dynamic and distributed deployment of the software needed to support a new Virtual Organization or a new Digital Library is characterised as follows:

- a set of software packages has to be installed in order to make the VO/DL working.
- a set of DILIGENT Hosting Nodes is available for installing packages.
- each package has some requirements on the DHN hosting it. The types of requirements that can be established on a DHN:

  - Property requirements (e.g. OS type == Linux)
  - Capacity requirements (e.g. CPU speed > 500MHz)
  - Consumption requirements (e.g. disk space > 30MB)
  - Software requirements (e.g. libX v2.4 has to be already installed)
- each DHN is characterised by its actual status as defined in Section 5.5.1.2 and derived from the view of the DHN provided by the HNM service (see Section 6.5.2.3).
- some packages have some 'uses' relationships with other packages meaning that after deployment, the corresponding software pieces will establish some kind of interaction. Although it is desirable to minimize network communication, the fulfilment of this constraint is not to be considered mandatory for a proposed deployment solution.
- some packages have some 'same-host' constraints meaning that the two involved packages have to be deployed on the same DHN. The fulfilment of this constraint has to be considered mandatory in a solution; its violation would lead to a faulty deployed system.

The problem is to find a mapping of packages onto DHNs such that:

- For each package, its requirements are satisfied by the identified DHN;
- Each DHN satisfies the aggregation of consumption requirements of software being hosted (e.g. the sum of disk space used by packages being installed on DHN X must not exceed the current available disk space);
- Where required, packages are deployed on the same hosts;
- The number of packages related through a 'uses' relationship deployed on different DHNs (the objective function) is minimised.

The Algorithm

The matching algorithm evaluates a deployment request for multiple packages against a set of DHNs descriptions and returns a mapping of packages onto hosts.

An exact algorithm would try every possible deployment and select the one that satisfies the constraints and minimises the number of 'uses' relationships violated. The complexity of this algorithm is $O(h^p)$ where h is the number of available hosting nodes and p is the number of packages to be deployed. As these number can be arbitrarily large, it is not typically feasible to evaluate all possible combinations. Instead, a greedy heuristic algorithm is used to find a suitable solution. It is composed of two phases:

- In the *grouping phase* packages are clustered together according to 'same-host' requirements. Packages in each group need to be deployed on the same host. For the sake of matching, packages in each group are represented by a new package whose requirements are the conjunction of requirements of individual packages in the group (consumption requirements are aggregated in the representative package). After this stage, no mandatory dependencies exist among packages.
- In the *assignment phase*, packages are incrementally assigned to DHNs. At each step, the package with more strict requirements is selected for deployment. This can be done by choosing the package with the lower number of *compatible DHNs* (a compatible DHN is a node satisfying all requirements of a package). Then the "best" assignment to a compatible host is selected (the best assignment is the one that minimises the objective function). The process repeats until no more packages have to be assigned to nodes.

Complexity Considerations

Let p be the number of packages being deployed, and h the number of available hosts.

The *grouping phase* consists in identifying connected subgraphs of a graph and requires O(V+E) where V is the number of vertices and E is the number of edges. The number of edges, in the case of a complete graph can be V^2; thus the complexity of the grouping phase is O(p^2).

In the *assignment phase*, the selection of the package requires O(p*h). The selection of the "best" node requires O(h*p) where the evaluation of the objective function requires O(p). Being O(h*p) the complexity of each step, the overall complexity is O(h*p^2)

Using the Keeper feedbacks

As described in the Use Case View section, deployment of a package may fail. After a failure the Keeper can notify the BMM about the event by signalling which packages have failed to install on which hosts. The BMM, in turn, can exploit this information to improve the quality of subsequent matching requests.

In particular, if the deployment fails because of few resources available, the most likely reason is that the status of the DHNs environment maintained by the BMM is not up to date. The action to be taken is to force a refresh of the status, and possibly to increase the notification frequency rate from the DIS.

It also may be the case that the failure is due to incompatibilities outside the scope of the model supported by the DILIGENT platform. In this case, the couple (p,h) is marked as 'bad' and node h is no longer considered for deploying package p.

5.5.1.2 Modeling the DHN status

To model the DHN status, the generic Host Model defined within the GLUE Schema Specification[8] is adopted. GLUE aims at defining an information model and mapping of the status information to concrete schemas for representing Grid resources. The following diagram shows the Host Model as from the GLUE Specification:

[8] GLUE Schema Specification version 1.2 Draft 8 available at http://infnforge.cnaf.infn.it/glueinfomodel/

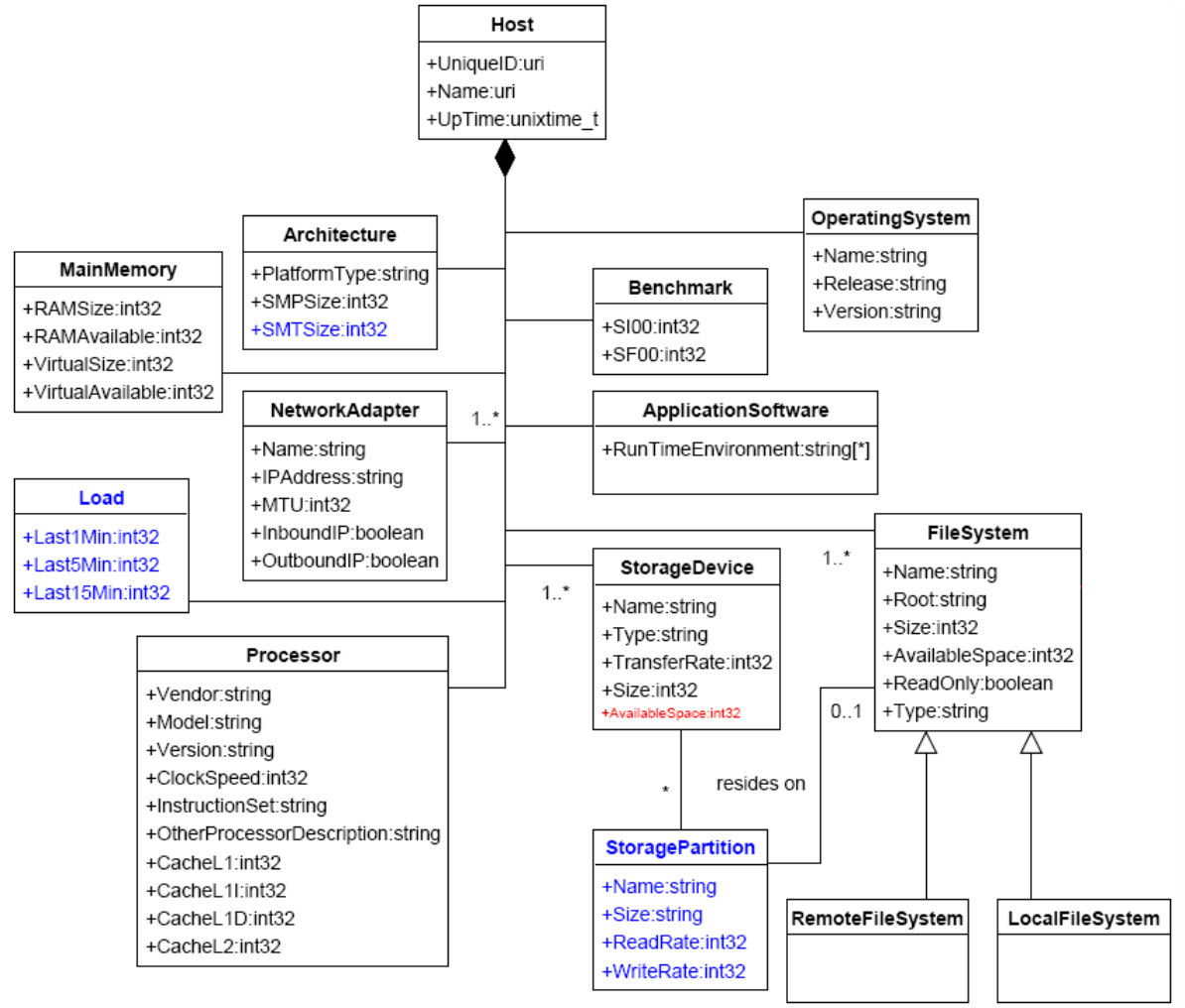

Figure 13. Generic Host model from GLUE Schema

The above model, however, lacks the description for installed DILIGENT packages (of course); this is indeed required to properly satisfy matchmaking requests. For this reason, the Host Model is enriched by modelling the set of packages currently installed. The model adopted for packages reflects the one given in Section 3.1.

The extended model is presented below:

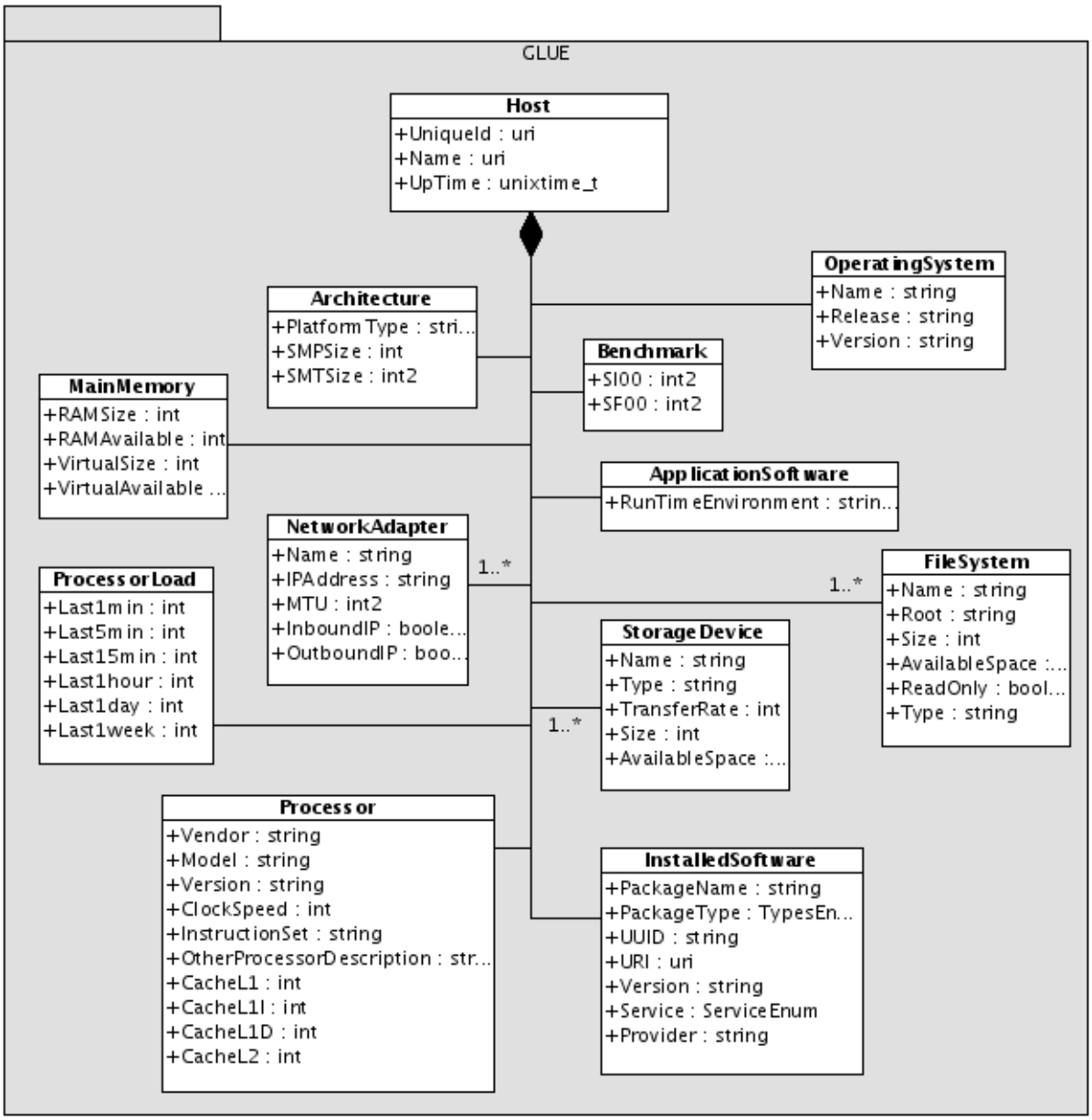

Figure 14. Package-extended GLUE Host Model

It can also be noticed that, for the ProcessorLoad category, last-hour, last-day and last-week average load properties are also maintained for each host. The reason for this extension is that VO/DL components are usually hosted on a DHN for a relatively long period of time. So, when deploying a component it would be useful to know long-term processor load instead of very short-term load as the last-min, last-five and last-fifteen measures are.

Also notice that these measurements are not published by the Host's HNM service, nor notified by the DIS; instead they are calculated by the monitoring subsystem in the *Processing and Filtering* phase.

5.5.1.3 Language for MatchMaking Requests

The Installable Unit Deployment Descriptor (IUDD) has been adopted to express instances of deployment problem being addressed by the BMM.

The IUDD Specification[9] defines an XML Schema allowing the description, in terms of requirements, of atomic software units as well as complex, multi-platform, heterogeneous solutions. A solution is a combination of software components addressing a particular user requirement. Software constituents of a solution can be grouped in a single IUDD; in addition, IUDD allows the description of the topology of targets onto which the solution has to be deployed.

The IUDD specification defines an *Installable Unit* as the fundamental building block or component from which solutions are composed. The installable unit is installed into a *Hosting Environment*, which is a container that can accept the artifacts of the installable unit.

The most important characterisation of installable units within an IUDD is represented by their requirements over the target hosting environment. The following check types are defined:

- Property checks for attributes, such as operating system type;
- Capacity checks for defining upper bounds of resources used, such as processor speed;
- Consumption checks for defining the amount of resources used, such as disk space;
- Installable unit checks for dependencies on specific, named installable units;
- Software checks for dependencies on software packages that might not be an installable unit.

5.5.1.4 Language for Solutions

[to be provided in the D.1.2.3]

5.5.2 Components

Two different components have been identified for the service:

- The MatchMaker (MM) service itself, grouping the *MatchMaking*, *Monitoring*, *ReplicaManagement* and *Catalogue* logical packages
- The BMM-API library, delivering an implementation of the *BMM-API* logical package

5.5.2.1 MatchMaker

5.5.2.1.1 State description

For the design of the *MatchMaker*, the singleton pattern has been adopted. In particular, for each VO (and thus for each DL), a *MatchMaker* service provides access to a single BMMResource maintaining an up-to-date view of the VO's DHNs status, and being responsible for matching activities.

As introduced in the Logical View Section, three alternative mechanisms are provided to persist the *BMMResource* state:

[9] The IUDD specification, previously submitted to W3C (http://www.w3.org/Submission/2004/SUBM-InstallableUnit-DD-20040712/), is currently being developed by the OASIS SDD Technical Committee under the new label of Solution Deployment Descriptor (SDD): http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sdd

- a file-based approach stores the current status of VO's DHNs in the local filesystem; for each DHN, an XML representation[10] of its status is stored in a different file.
- a DBMS-based approach exploits a relational database as persistence support for VO's DHNs status.
- a native XML database is adopted to store XML representations of DHNs status.

5.5.2.1.2 Operations

The MatchMaker exposes three operations to clients:

- **findOptimalAllocation**(deploymentRequest)::deploymentSolution – The input parameter is a (list of) IUDD describing the set of packages being installed along with their requirements and dependencies. The output of this operation is a deployment solution mapping packages listed in the request to VO's DHNs.
- **feedback**(packageId, hostId, message) – This operation is used to notify when a failure occurred while deploying a package on a node. The notification may include the reason of the failure to better tune the matching phase.
- **downloadFeedbacks**()::feedbackList – this operation is used by MatchMaker replicas to restore their status after a MatchMaker restart/crash.

5.5.2.1.3 Profile description

The profile of each MatchMaker registered in the DIS includes a reference to the VO it belongs to. This is mainly intended for the Keeper (actually for the BMM-API library it uses) to discover MatchMaker replicas for a VO.

The security profile of the MatchMaker is the following:

- **MMSecurityDescriptor**: default
  - **Authorization Chain**: only VO-level authorization is performed. For each VO, only the Keeper Service is enabled to use the service. The downloadFeedbacks() operation has no associated authorization policy.
  - **Default Settings**
    - **Identity Scenario**: Service
    - **Authentication Methods**: GSISecureMessage (integrity & privacy)

5.5.2.1.4 Status description

[to be provided in D1.2.3]

5.5.2.1.5 Dependencies & Requirements

The MatchMaker service relies upon the following entities:

DILIGENT services

DILIGENT Information System – the DIS is exploited by the MM in order to maintain an up-to-date status of the VO's DHNs environment status. In particular, the DIS-HLSClient library is used:

- by the Monitoring subsystem to subscribe for notifications about DHN status changes

[10] An XML schema for the original GLUE Host Model can be found at http://viewcvs.globus.org/viewcvs.cgi/playground/java/glue/schema/base/glue/ce.xsd?rev=HEAD&content-type=text/vnd.viewcvs-markup. The adopted schema will take into account the extensions to the GLUE Host Model proposed in section X.

- by the ReplicaManagement subsystem to discover MM replicas for the same VO
- by the BMM-API library to discover a replica to contact after a Keeper request

5.5.2.2 BMM-API

5.5.2.2.1 Description

The BMM-API library provides local access to BMM functionalities, avoiding the need to select and contact a MatchMaker Service replica. For this, the BMM-API interacts with the DIS-HLSClient class in order to discover the location of a BMM.

5.5.2.2.2 Usage

[to be provided in the D.1.2.3]

5.5.3 Deployment scenario(s)

[to be provided in the D.1.2.3]

6 KEEPER SERVICE

6.1 Introduction

According to the DoW, the role of the Keeper Service in the DILIGENT system is the following:

"The Keeper service has to instantiate the set of services belonging to a VDL and to manage them assuring the characteristics of QoS required by the VDL Definition criteria".

This definition uses the term "VDL" to identify the set of resources selected by the VDL Generator Service forming a VDL. To be compliant with the terminology adopted in the VDL Generation Service design (Section 8), we use the term "VDL" for the design phase of a Digital Library (where the DL is *virtual*), and the term "DL" starting from the creation phase performed by the Keeper. Thus, the term VDL in this section is used referring the VDL Definition criteria elaborated by the VDL Generator Service and used as starting point for the creation of a concrete DL.

Starting from the above definition, the Keeper is primarily responsible for the creation of a new VDL by instantiating its resources (i.e. Running instances of its services) and by authorizing its users. It must also guarantee the overall set of the VDL functionalities at any time by dynamically relocating resources and periodically checking their status. For instance, if the number of queries executed in a DL is higher than an established threshold and causes an unacceptable execution time with respect to the QoS requested by the user, the Keeper creates new Search and/or Index services, configures them to be usable in the DL, and improving the performance of the search service should then solve the problem.

Along the design time, the tasks of the Keeper are increased, and it is now also in charge of the deployment of all the components needed for the management of a new VO, and of the registration of new DILIGENT resources.

6.1.1 Main tasks

The main task assigned to the Keeper is the dynamic deployment of a service to create a new running instance of that service on a specified node. This activity is performed in different cases:

- when a new DL is set up,
- when it is necessary to maintain the level of QoS requested for a DL,
- when a critical fault occurs in a running instance or a node,
- when a new VO is created,
- when a new DILIGENT resource registration event is raised by the DIS,
- when a new node joins the DILIGENT VO.

In particular, a number of other activities must be performed by the Keeper when one of the previous events occurs.

In particular, when a new DL is designed, it has to:

- manage the DL definition criteria,
- interact with the Broker & Matchmaker to identify the appropriate nodes where to deploy the identified services,

- manage services dependencies, requirements, and QoS parameters[11].

During the DL lifetime, it has to:

- coordinate and disseminate the operational context that transforms this set of distributed DILIGENT resources into a single application. In the DILIGENT terminology, this context is named DL Map and, basically, it specifies the DL resources locations and their configuration. Any other dynamic information about a resource (e.g. its status) is maintained and disseminated by the Information Service.
- monitor a DL, and:
    - create new Running Instances in order to maintain the required QoS;
    - create new Running Instances after a fault;
    - respond to new DILIGENT resource registrations in the DIS.

When a new VO is created by the DVOS, it has to:

- identify the mandatory services to deploy for the management of the new VO

When a new node joins the DILIGENT VO, it has to:

- identify the mandatory services to deploy on the node for its management

6.1.2 DHNs and Packages

Two basic concepts, upon which the Keeper design is based, have been already introduced:

- Software Package
- DHN (DILIGENT Hosting Node)

Software Packages, introduced in Section 3.1, are single files, compliant with the Package Model, that contain the files to be installed, along with rules describing software/packages dependencies, deployment instructions, etc.

A *DHN* is a network node able to host DILIGENT services and related components. In order to become a DHN, a node must have installed the following software:

a. a hosting environment where DILIGENT services can be deployed;
b. a component of the Keeper that collaborates in the local deployment of packages.

The node must also be configured to join the DILIGENT infrastructure and the appropriate VOs.

These elements represent the minimal software that must be present on a DHN. Starting from them, any other DILIGENT software can be dynamically deployed.

The DHNs and Packages represent the foundation that makes it possible to move from quite static environments to the highly dynamic one we have in DILIGENT. Using them, resources can be created and moved at any time[12] and their consumers do not have to deal with this behaviour since the Collective Layer services guarantee a transparent and continuously up-to-date dissemination of the information.

Once we have clarified the meaning of these two concepts, we can reformulate the main task of the Keeper in the following manner:

[11] Actually, the Keeper service relies on the functionalities offered by the BMM service to provide these tasks.
[12] As explained later, this operation is performed guaranteeing a safety session termination.

"to provide all the functionalities that allow to handle software packages within any DILIGENT VOs and to deploy them on the appropriate DHNs when one of the event reported in section 6.1.1 occurs".

6.2 Use-Case View

Starting from the D1.1.1 document and according to the role reported in the previous section, we identify the following list of high-level functionalities that are related with the Keeper Service:

4.2 DLs Management

- 4.2.14 Create a DL
- 4.2.15 Check DL Definition
- 4.2.16 Analyze available resources
- 4.2.17 Include DL Users
- 4.2.18 Create DL Resources
- 4.2.19 Generate Web Portal
- 4.2.20 Maintain a DL
- 4.2.21 DL Resource Monitoring
- 4.2.22 Report DL status
- 4.2.23 Update a DL
- 4.2.24 Remove a DL
- 4.2.25 Remove DL Resources

4.3 Resources Management

- 4.3.9 Manage a Resource in a DL
- 4.3.10 Add a Resource to a DL
- 4.3.11 Create a DL Resource
- 4.3.13 Configure (DL) Resource
- 4.3.14 Update a DL Resource
- 4.3.15 Remove a DL Resource

Moreover, the Keeper uses the following functionalities that must be provided by other DILIGENT services:

4.3 Resources Management

- 4.3.2 Register a Resource
- 4.3.5 Store Resource Profile
- 4.3.6 Remove Resource Profile
- 4.3.12 Find Optimal Allocation
- 4.3.17 Get Available Resources
- 4.3.20 Monitor a Resource

4.6 Notifications Management

- 4.6.2 Notify Role

4.4 VOs Management

- 4.4.2 Create a VO
- 4.4.3 Add a Resource to a VO

- 4.4.4 Edit Resource Policy
- 4.4.6 Add a User to a VO
- 4.4.13 Remove a Resource from a VO

4.2 DLs Management

- 4.2.10 Preserve Content

Two human actors are directly related with the Keeper Service:

- DL Manager
- Resource Manager

The DL Manager is the supervisor of the whole work performed by the service. It is in charge of receiving the notification of a new DL creation from the VDL Generator Service (on behalf of DL Designer) and to start the DL creation process.

On the other hand, a Resource Manager must configure a network node in order for it to be able to become a DILIGENT Hosting Node (in short, DHN), i.e. a node able to host DILIGENT services.

Moreover the huge part of the work is performed during the entire DL life-cycle: after the DL creation, the Keeper must keep the whole set of services and resources up and running in a consistent way with respect to the expected functionalities and the QoS specified by the DL Designer.

By analyzing these functionalities and taking care of the above considerations, the following use-case view presents the significant functionality of the service grouped by subsystems. Moreover the dependencies with other DILIGENT components provided by other services are also highlighted.

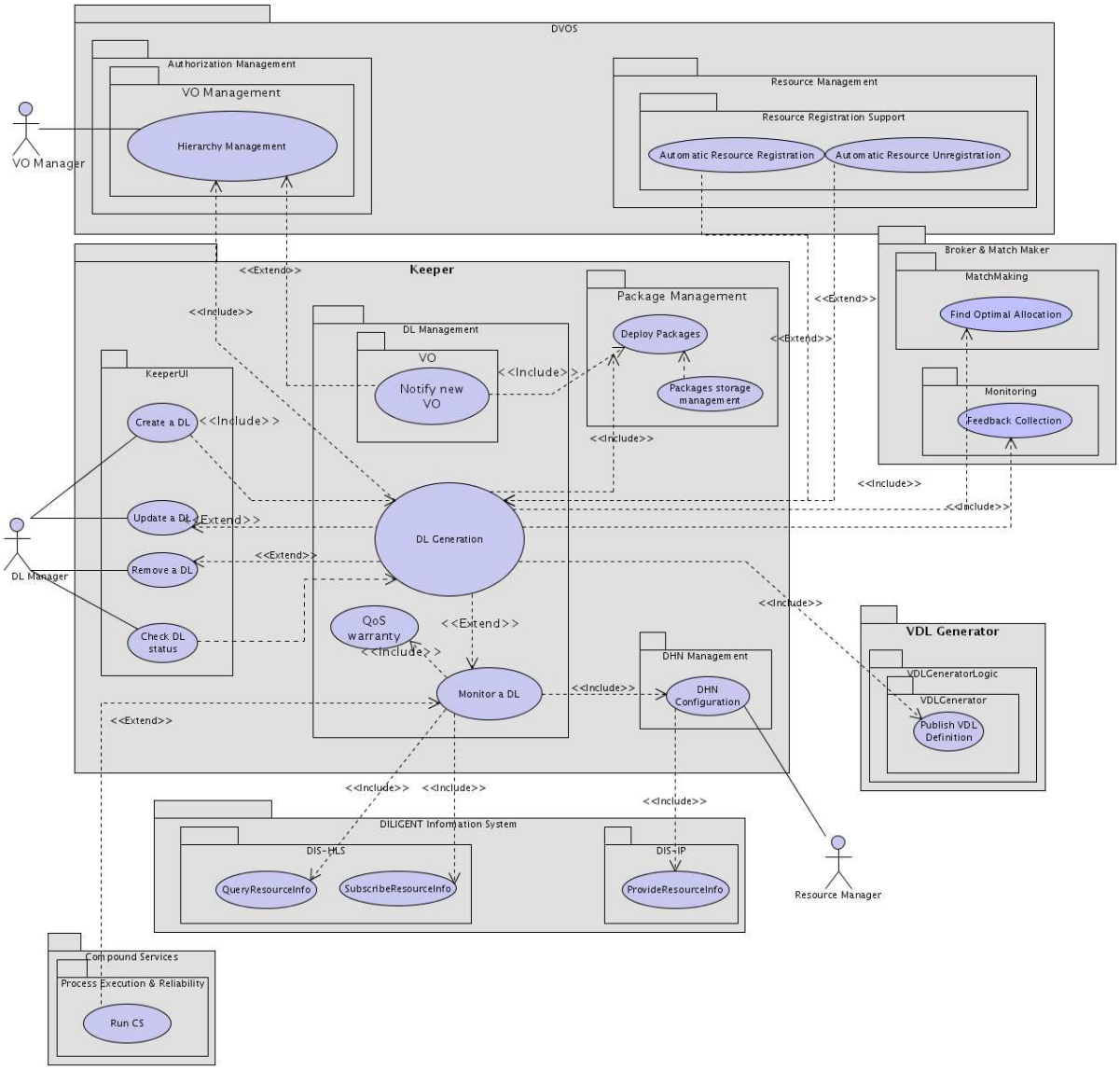

Figure 15. Keeper Service - Use-Case View

The sub-packages inside the main Keeper package identify a preliminary decomposition of the service structure that is used as a basis to build the software architecture. Hereafter we present an overview of each of them. Details about all these issues are provided in Section 6.5.

KeeperUI package

This package groups use-cases that model actions that the DL Manager can directly access, i.e. it lists the functionalities that can be accessed through the graphical user interface of the service.

- **Create a DL** – After receipt of the notification from the VDL Generator, the DL Manager uses this functionality to start the process of generating a new DL.
- **Update a DL** – Through this functionality, the DL Manager can change the DL service composition and/or distribution, or notify the DL Designer about an unwieldy problem met in the DL that requires a DL redesign. These actions are undertaken

after the receipt of a notification from the DL Monitor functionality or a Check DL status report.

- **Remove a DL** – A DL Manager can remove a DL as a consequence of a request from the DL Designer or when the DL lifetime expires.
- **Check DL status** – This functionality reports to the DL Manager the current status of the resources that compose the DL.
- **Enforce Resource Policy –** This functionality represents the local authorization policies according to the design of the Authorization Management (Section 7.2).

DL Management package

The second group of functionalities is the DL Management that is in charge of managing the service instances that form a DL.

- **DL Generation** – This functionality groups a huge set of core functionalities of the Keeper service. DL Generation works in conjunction with the Broker & Matchmaker service to dynamically deploy the software packages that compose a new DL.
- **Monitor a DL** – After its creation, a DL must be maintained and monitored in a consistent way. This functionality periodically checks and manages the set of DILIGENT resources that form a DL.
- **QoS warranty** – This functionality models the activities that the Keeper performs to ensure that the requested QoS level of the DL is maintained

DL Map package

In our experiences with distributed architectures for Digital Library applications we have always pointed out the need of a DL Map. A DL Map is a container of all the information needed to the Keeper to manage the group of services that form the DL. It defines a means by which an application context in a distributed environment can be created.

The DL Map package includes functionalities that make concrete these goals.

- **DL Map Management** – This use-case groups all the functionalities that allow for building and manipulating the DL Map.

Package Management package

In order to create new service instances we need to *(i)* handle the software to be deployed and *(ii)* install the service on a DHN. The two use-cases of the Package Deployment package do these activities:

- **Packages storage management** – A service is usually composed by one or more software packages. The packages are uploaded through the Request Resource Registration functionality provided by the DVOS service (Section 7.5) and then they are stored in the system via this Keeper functionality.
- **Deploy services** – This use-case models the deployment of a package on a DHN node.

DHN Management package

To complete the scenario, we need functionalities that abstract and make accessible the DHN configuration to the services hosted on the DHN. This is the goal of the DHN Management package.

- **DHN Configuration –** This use-case models the functionalities that allow to manage and expose to the DIS and to the local services:
  - the manual DHN configuration set by the Resource Manager (as DHN owner)
  - the ongoing configuration of the installed packages on the DHN

VO package

This package provides a functionality that is not foreseen in the D1.1.1 document because it represents a requirement that just showed up during this service specification phase. When a new VO is created, some basic packages must be deployed on the DHNs of the new VO in order to manage it; the Keeper is the obvious candidate to do that. The **Notify New VO** functionality models this behaviour.

6.3 Logical View

By analyzing the use cases of the use-case view, the logical decomposition of the Keeper Service depicted in Figure 16 results. In this logical vision, each sub-package presented in the use-case view is expanded in its logical classes and relationships among these classes are also shown except the VO package. This package is represented by the VO Deployer class that is part of the DL Management package. This choice is mainly due to the fact that its logic (the deployment of software packages) is shared with the classes of that package.

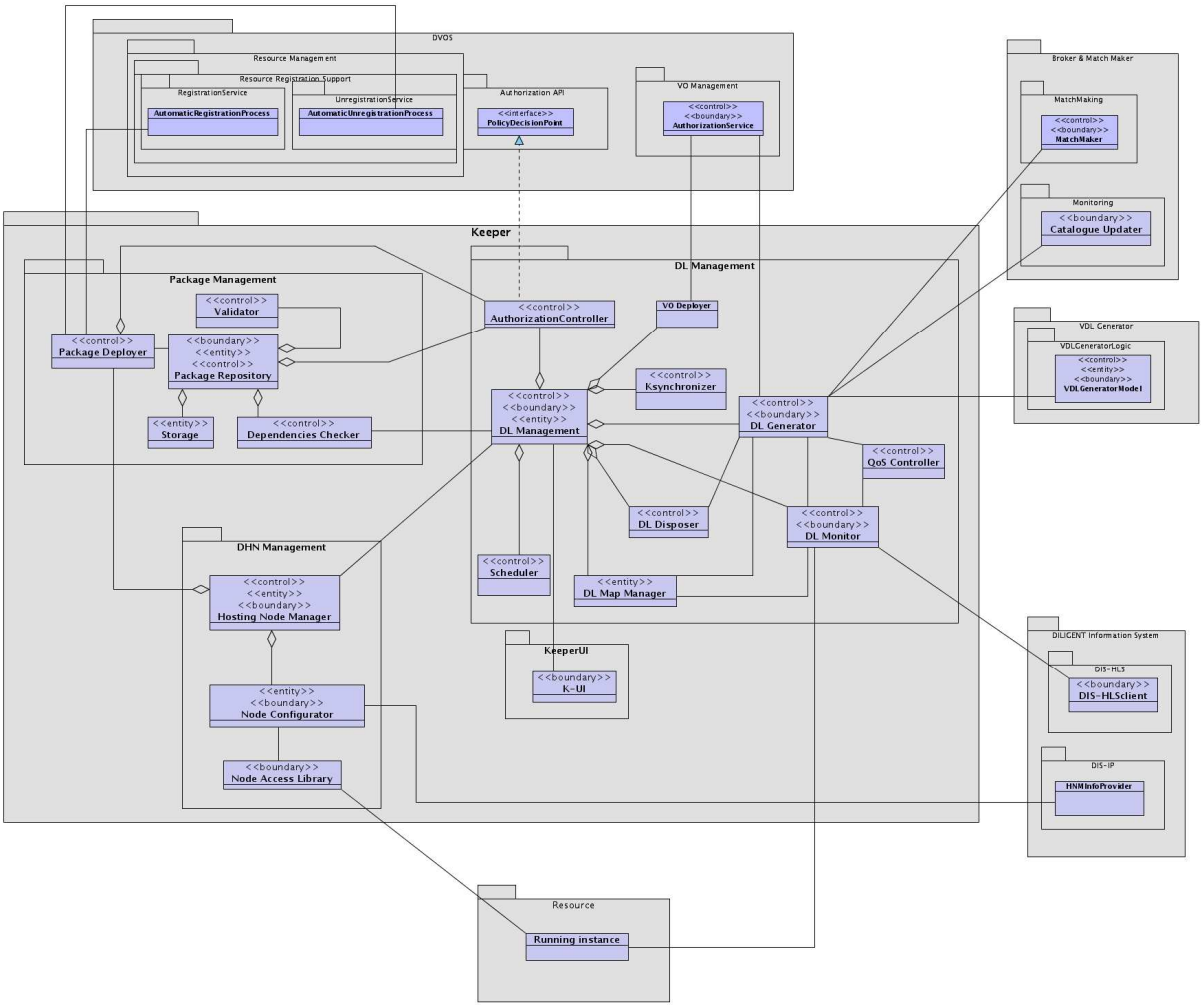

Figure 16. Keeper Service - Logical View

A short introduction of each package is provided below. A more detailed description of each class and their interactions with other classes and components are presented in Section 6.5 during the description of the related components.

DL Management package

The DL Management includes classes that manage the creation, monitoring and disposal of service instances and any other generic software package registered in DILIGENT. As stated in the Use-Case View section, they are performed when various events occur. These classes work in conjunction with the DHN Management and Package Management packages to perform the deployment process. The monitoring activities include the guarantee of the QoS requested for the DL by taking decisions about the DL topology. Finally, they manage and maintain an up-to-date version of the DL Map used to discover information (mainly DL specific data, the location and configuration) about services that form the DL.

These classes make use of a number of other classes provided by the following DILIGENT services:

- Broker & MatchMaker – to discover new DHNs where services can be deployed
- DIS – to obtain information about services and resources
- VDLGenerator – to establish the list of services that compose a DL
- Dynamic VO Support – to create the new Sub-VO related to a new DL and to add resources and users to it; to register new packages, and to deploy mandatory packages for a new VO
- Generic DILIGENT Service – to query its status and accept notifications about its faults

KeeperUI package

The KeeperUI classes model the interactions with the DL Manager. It contains classes that allow to start the process of DL generation, to update an existing DL, and to dispose a DL. Moreover a DL Manager can interact with the DL Management package classes in order to obtain a report of the DL status at any time.

DHN Management package

These classes are involved in the deployment of new software packages on the DHNs and in the management of the local configuration of the DHNs. Their role is to accept the instructions of the DL Management classes and to start the deployment process using the Package Management classes.

Package Management package

The Package Management classes model the storage of software packages, the action of moving them on the appropriate DHNs and the management of their deployment process in conjunction with the DHN Management classes.

6.4 Deployment View

By analysing the logical view depicted in Figure 16, the Keeper has been decomposed in four components that can be hosted in different networked locations:

- Package Repository
- K-UI
- DL Management
- Hosting Node Manager

From the DILIGENT Resources Model perspective, all these components are distributed as WSRFService packages with the exception of the K-UI that is a Portlet package (see 6.5.2.4).

In addition to these major components, other packages are distributed separately as Library packages; they are the Stub Libraries used by these components or by other services to interact with: i) the Package Repository, ii) the DL Management, and iii) the Hosting Node Manager.

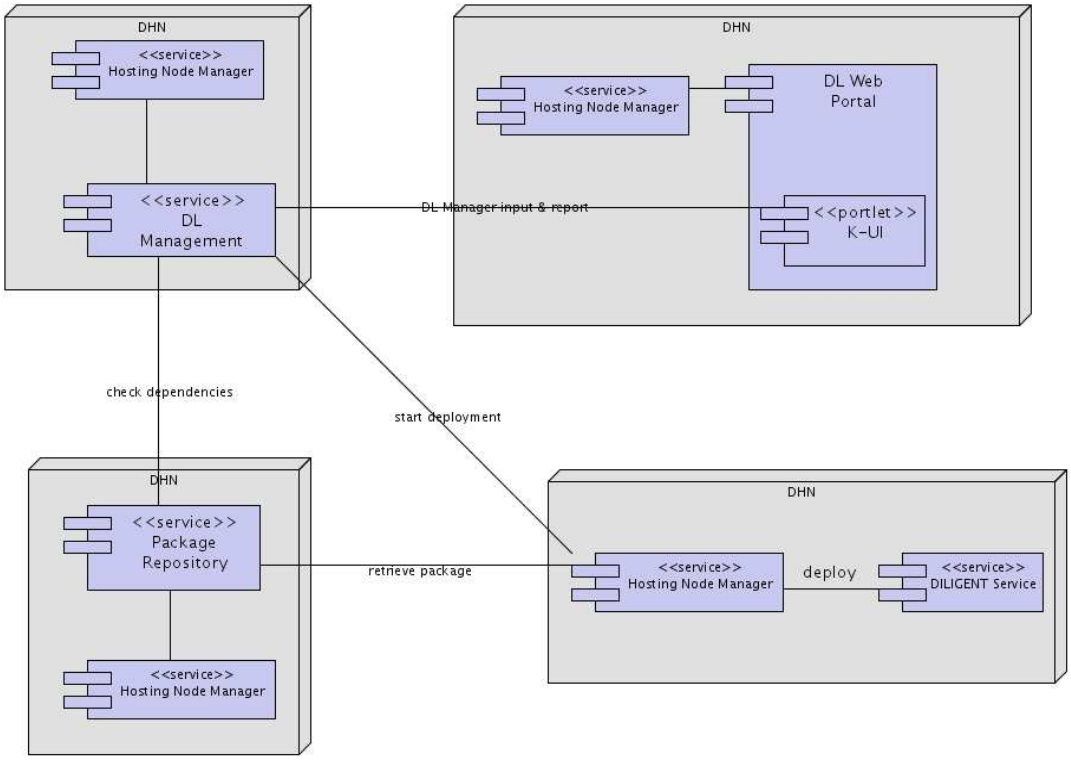

Figure 17. Keeper Service - Deployment View

The **DL Management** is a WSRF service in charge to start the deployment process of packages and to respond to events that occur during the DL life-cycle. There is at least one DL Management for each DL and for each VO.

The **Hosting Node Manager** is a WSRF service that must reside on each networked node that aims at hosting DILIGENT services. It is in charge to deploy new packages on the node following the statements of the DL Management.

The **Package Repository** is a WSRF service that handles all installable software.

Finally, the **K-UI** portlet is the graphical user interface integrated in the DL portal that permits the execution of some administrative tasks and to access to a report of the DL status.

The Stub Libraries released within the Keeper are not depicted in the diagram since they are just small software packages that allow to interact with each service in a simple way.

Section 6.5 provides a detailed description of each component.

6.5 Service design

6.5.1 Design considerations

As DILIGENT follows the SOA paradigm, it is clearly composed by services that must be accessible via a network interface. It is not a good approach to build such software from scratch since in the web development area there exists a great number of sophisticated open source solutions dedicated to services hosting. These solutions are usually referred to by the term *hosting environments*. The adoption of a specific hosting environment influences the way in which services are designed, developed, packaged and distributed. Having one unique environment simplifies the distribution of the DILIGENT Services since it makes it possible to install them in the same way without providing an ad hoc solution for each service.

The selected hosting environment chosen for deployment of the services is the *Java WS Core* developed by the Globus Alliance. This container provides a complete implementation of the WSRF and WS-Notification working draft specifications plus WS-Addressing and WS-Security support based on the Axis Web Services engine developed by the Apache Foundation.

Of course, the Keeper design has been done to be as platform independent as possible even if it takes into account the WSRF standards and technologies.

The Package Model

So far, we have stated that the Keeper manages software packages. Usually, a package is defined as a software component prepared for integration into a suitable computer system. Moving into the DILIGENT context, packages are "pieces of software" able to be deployed on a DHN; a lot of information is needed to manage them correctly. In order to organize this information we have introduced the Package Model (Figure 3) against which the uploaded software packages are validated. If a package respects the logical structure imposed by this model it is accepted as new DILIGENT software.

As presented in Figure 3, there are four kinds of package: WSRFService, GridJob, Portlet and Library. Each of them has to conform to a particular deployable format reported in Table 1:

Package	Deployable format
WSRFService	Grid Archive (GAR)
Portlet	Web Archive (WAR)
Library	Java Archive (JAR)
GridJob	gLite job

Table 1. Software package formats

Each package is also equipped with a set of additional information that drive the Keeper in the deployment process. They include:

- Requirements about the DHN on which the package is installed that can be
  - Requirements against the hardware and software installations (e.g., against the Host model presented in Section 5.5.1.2),
  - Requirements against other DILIGENT software packages installed;

- Requirements against the DL, such as "if this package is deployed the package X needs to be also deployed within the DL" or "this package needs that another packages configured with the param X=Y is deployed within the DL";
- Scripts that can be executed before the package deployment to prepare the runtime environment for it (checking for third parties installed software, setting of environment variables, modification of the filesystem, etc.) and to clean it after service removal.
- Qos parameters such as:
    - Proximity with other packages,
    - Desirable hardware and network requirements,
    - Maximum number of requests in a unit of time managed by the overall service or a particular service invocation (in the case of a WSRFService package),
    - [to be detailed in D1.2.3].

The complete list of this optional information is foreseen for the D1.2.3 since they depend on the forthcoming requirements from WP1.3-WP1.6 services design.

In the Package Model, some information have the goal of forcing the deployment of a package when a particular event occurs – even if it is not explicitly requested by anyone and it does not have any dependencies with other packages. This information is represented by the following three flags:

- *VOMandatory* – the package is deployed when a new VO is created (i.e. a Broker & MatchMaker or an AuthorizationResource)
- *DLMandatory* – the package is deployed when a new DL is created (e.g. one or more packages must always be deployed in a DL) and becomes part of the DL
- *DHNMandatory* – the package is deployed on a DHN when that DHN is added to the DILIGENT VO (e.g. the package must be deployed on all DHNs)

6.5.2 Components

In Figure 17 we have depicted the decomposition of the Keeper services in 4 components. We are confident that this architectural design is the right balance between an optimization of the distribution of the components tasks and the reduction of the overhead introduced by the network and SOAP communications. We introduce this architecture starting from the following considerations:

- there is the need to have "something" on each node to perform local deployment activities;
- the software packages files have to be stored somewhere and they are not necessarily DL specific
- a main component in charge to handle that DL is needed
- the DL Manager must interact in some way with the Keeper

These basic issues drive us to the detailed design that we discuss in the following sections.

6.5.2.1 DL Management

DL Management is the component in charge to build and manage a DL. It is composed by several modules (also known as "subsystems" in UML terminology). Figure 18 introduces these modules and their interactions with other Keeper components on other nodes.

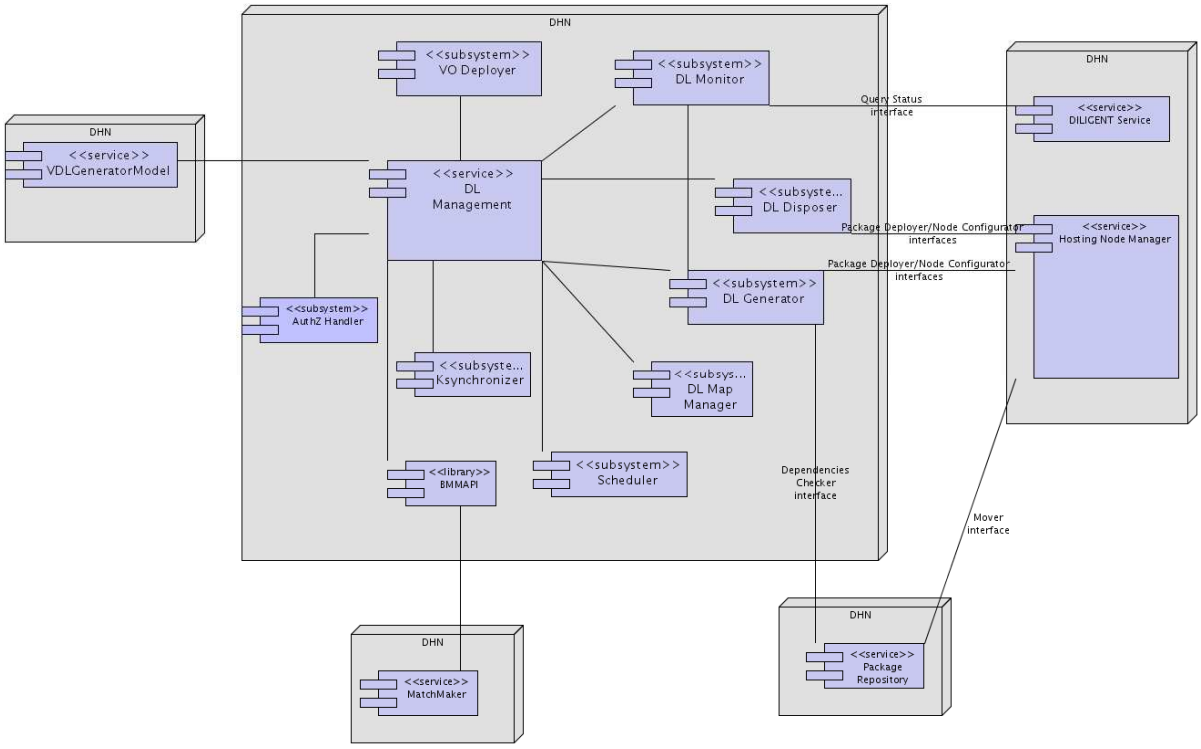

Figure 18. DL Management Node - Deployment Diagram

Depending on the assigned scope, a DL Management has a different behavior. There are two possible scopes: the *VO scope* and the *DL scope.* In the first case the goal is to manage packages within VOs as well as the DL Management instances of the DLs belonging to the VOs. In the second case the goal is to manage packages for a specific DL.

DL Management with "VO scope"

When a DL Management has a "VO scope", its main task is to ensure a correct deployment of the packages that allows to manage the assigned VO (Figure 19). At the moment of writing this deliverable, these packages are: the DL Management itself, a MatchMaker and an AuthorizationService.

In this case, the DL Management is also responsible for receiving new DL requests and starting the process that leads to the generation of a DL (see Figure 20).

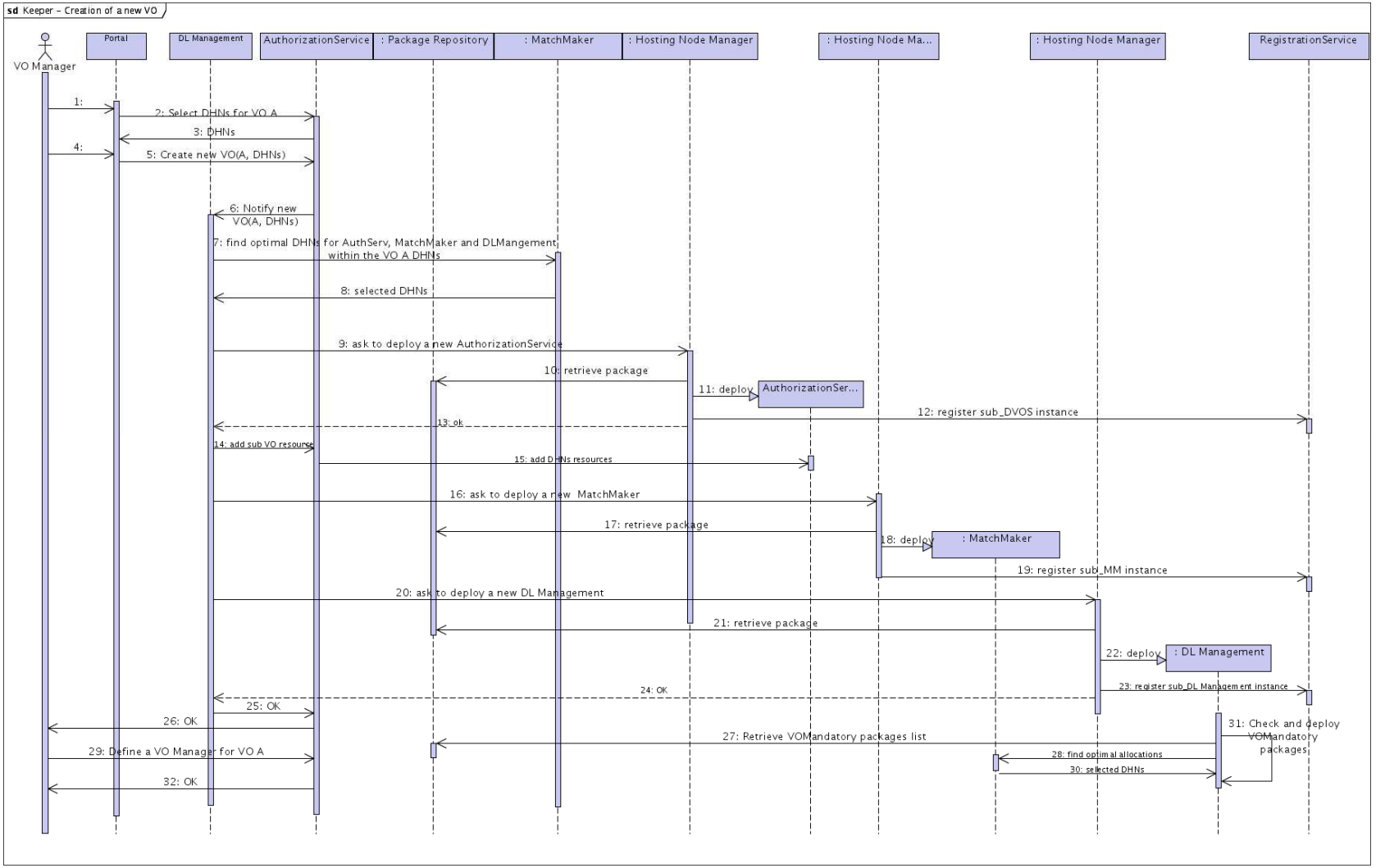

Figure 19. Keeper Service - The creation of a new VO

A DL management with a VO scope is invoked when one of the two events occurs:

- *when a VO Manager decides to create a new VO*: the AuthorizationService (see Section 7.2) service notifies the DL Management (step 6), which starts the deployment process of the three components needed to manage the rising VO: a new AuthorizationService, a new MatchMaker and a new DL Management (steps 7-25). Then, the new DL Management (VO scoped too) deploys (step 30) on the DHNs of the new VO the packages that are labeled as "VOMandatory" (see Package Repository Section). Of course, these deployment operations are performed in conjunction with the MatchMaker service that selects the target DHNs.
- *when a DL Manager decides to create a new DL (following the request of a DL Designer)*: see next diagrams and the related description.

DL Management with "DL scope"

Figure 20 shows how the DL Management is involved in the process of creation of a new DL.

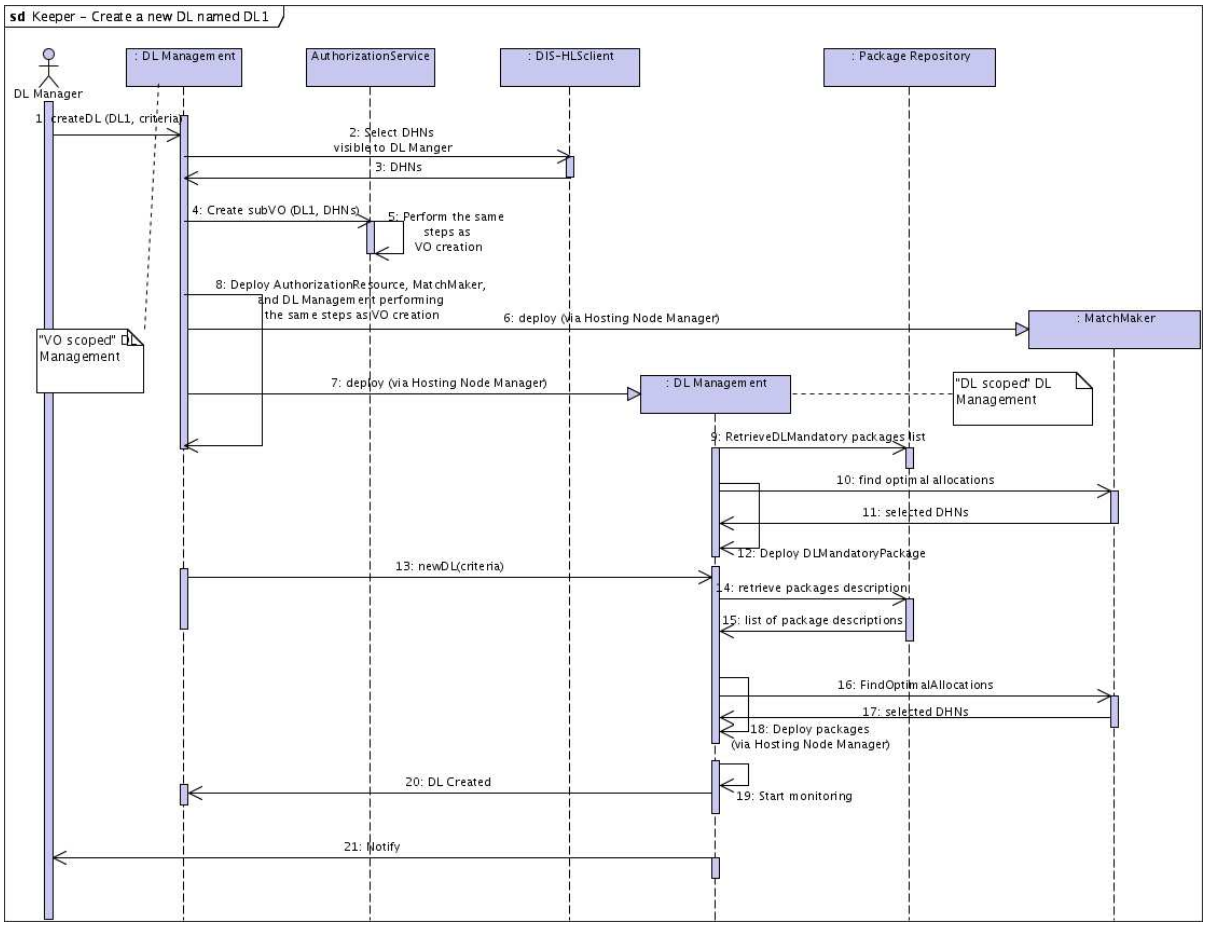

Figure 20. Keeper Service - Creation of a DL

A "DL scoped" DL Management is created by a parent DL Management with a "VO scope" that receives the request of a new DL creation (step 1) from a DL Manager. A new sub-VO is created in collaboration with the DVOS and DIS services (steps 2-5) that includes by default all the DHNs visible to the DL Manager (or a subset, if specified in the DL criteria). Then, the parent DL Management deploys a new DVOS, a new MatchMaker and a new DL Management for the sub-VO (steps 6-8). As first task, the deployed DL Management deploys the "DLMandatory" packages (see Package Repository section). Finally, it receives the definition criteria from the parent DL Management (retrieved from the VDL Definitions

Repository, see Section 8.5.2.4) and starts the creation process of the new DL and the subsequent monitor activities (steps 11-18).

In the following section, we provide a description of each subsystem of the DL Management.

DL Generator subsystem

Each time a new DL has to be created, the DL Generator examines the DL definition criteria, interacts with the Packages Repository and builds the list of packages that must be deployed in order to run the new DL. These deployable resources can be services, service components, shared or stub libraries, portlets or tasks (see Section 3.1). Once the list is complete, it interacts with the Broker & MatchMaker service (by using the IUDD information about the packages, see the Broker & MatchMaker section for further details) in order to obtain the DHN nodes on which to deploy these resources.

We have to point out that a rising DL can (if it has the rights to do so) also join pre-existing resources (e.g. archives, collections); in this case the DL Generator must be able to support this task in conjunction with the service instances that manage the authorization and authentication of such resources.

Major outcomes of DL Generator activities are:

- the creation of the DL sub-VO
- the creation of all DL Resources
- the inclusion of all DL Users
- the production of the DL Map

DL Monitor subsystem

After a DL is created, a major activity of the DL Management component is to monitor the DL running instances of the services that form the DL in order to avoid/repair any damage and to maintain the level of QoS requested by the DL Designer.

Monitoring a DL means:

- periodically pinging the running instances,
- receiving and managing reports from the running instances,
- checking the workload of each running instance in order to create new ones to balance the work and improve performances.

Pinging activity can be performed only on those instances that expose an interface that can be queried in order to investigate the service status. To make the DL Monitor aware of the existence of this interface, the "DisposeInterfaceSupport" flag in the WSRFService package (see Package Repository section) must be set to a true value.

When a service instance fails the DL Monitor asks to the DL Generator to deploy a new instance of that service or, if this is not possible (because the service is not deployable on the fly), it reports the error to the DL Manager that decides what to do (contact the DL Designer to redesign the DL or change him/herself the DL).

DL Disposer subsystem

This module manages the removal process of a DL when the DL Manager decides to remove it or when the predefined DL lifetime expires. This process has the goal to deallocate and release the infrastructure resources used by the DL.

Starting from the DL Map, it communicates with the Hosting Node Managers on the various DHNs that host DL services and ask for the undeployment (or deactivation) of the packages related to the DL, if they are not shared with other DLs. Then it is in charge to remove the DL sub-VO (by interacting with the DVOS service) and to ask to the local Hosting Node Manager the deallocation of the DL Management component.

Scheduler subsystem

This module is just a simple and internal module that periodically executes administrative tasks. It also executes CSs (by using a CS Runtime component of the Process Management area) responding to particular events that occur. At the moment of writing this document, planned events to respond are related to changes in the information maintained by the DILIGENT Information Service.

Map Manager subsystem

The Map Manager is used by other modules (mainly by the DL Generator and DL Monitor) in order to add/delete/update entries in the DL Map. It offers the possibility to subscribe to notifications about changes in the map.

[to be detailed in D1.2.3]

KSynchronizer subsystem

A DL Management instance is a vital component of the entire DILIGENT application. Therefore, DL Management needs to be replicated. Once a DL Management with a "DL scope" is deployed (it is the master), as the very first step, the new instance must deploy other instances (representing the slaves; their number depends on the level of reliability requested to the DL in the definition criteria and on the DHNs available) and assign them a priority. If the master DL Management goes down the slave instance with the highest priority becomes the new master one and deploys another slave replica.

The role of the KSynchronizer is to assure the synchronization among different DL Management instances.

VO Deployer subsystem

The VO Deployer plays its role in a "VO scoped" DL Management. It receives notifications from the DVOS service about the creation of a new VO. Its role is to deploy on the DHNs of the new VO the three mandatory Collective Layer components (DL Management, MatchMaker and DVOS) that assure a correct management of the new VO and any other "VOManadatory" labelled package (see Package Repository section for further details).

6.5.2.1.1 State description

Depending on the scope, the component can have two different states:

- **VO scope**: for each managed VO it creates a new WS-Resource. The WS-Resource properties documents of these WS-Resources report very few information (just those needed to identify the WS-Resource associates to a particular VO) about the VOs since they are already described by the DVOS services;
- **DL scope**: the service manages only one DL per service instance, so, in this case it creates only one WS-Resource per instance. The associated WS-Resource properties

document reports some information about the DL (creation time, DL Manager, etc.) to be used by other DL Managements.

6.5.2.1.2 Operations

DL Management with "VO Scope"

- **NotifyNewVO** (input: VO data, DL Manager)
- **NotifyDisposeVO** (input: VO id)
- **CreateDL**(input: VDL definition criteria) – invoked by the K-UI to send inputs from DL Manager to a VO scoped DL Management

DL Management with "DL Scope"

- **DeployDLPackages** (input: VDL definition criteria) – invoked by the parent DL Management to start the deployment process of the new DL packages
- **ReportFeedback**(input: WSRFServiceReport) – invoked by Running Instances to report failures or QoS relevant facts
- **ReportFeedbackFromDHN**(input: DHNReport) – invoked by the Hosting Node Managers to report issues about a DHN
- **DisposeDL**()– invoked by the K-UI to send inputs from DL Manager to a DL scoped DL Management
- **UpdateDL**(input: new criteria) – invoked by the K-UI to send inputs from DL Manager to a DL scoped DL Management
- **KSynchState**(input: DL Management state) – invoked by the DL Management with the highest priority to notify its state to its replicas
- **SetDLManPriority**(input: new priority) – invoked by the DL Management with the highest priority to set the priority of its replicas
- **RunCSForEvent**(input: CS, event) – any Running Instance can ask the DL Management to execute CS when a change occurs in an information stored in the DIS.

[to be detailed in D1.2.3]

6.5.2.1.3 Profile description

[to be provided in D1.2.3]

6.5.2.1.4 Status description

[to be provided in D1.2.3]

6.5.2.1.5 Dependencies and Requirements

The DL Management relies upon the following entities:

DILIGENT services

DL Management interacts with the *DLGeneratorModel* service (a component of the VDLGenerator) in order to retrieve the VDL Definition criteria upon which the new VDL will be created. These criteria contains:

- A list of Resources (Archives and Services);
- The configuration of each resource;
- A list of invited users/groups.

DL Management interacts with the *Package Repository* in order to retrieve the dependencies among the different packages.

DL Management interacts with the *BMM-API* (a client library of the MatchMaker service, see Section 5.5.2.2) via IUDD to obtain the list of DHNs where the packages are deployed. During this interaction, the entire list of the packages that will compose the rising DL is passed to this service that replies with a list of possible deployment solutions. Each solution is a data structure reporting, for each package, a DHN that satisfies the requirements associated with the package at the package registration time.

DL Management also reports any deployment error to the MatchMaker service.

DL Management interacts with the Dynamic VO Support in order to create a new sub-VO and add the DL Resources and DL Users to it. It also implements the necessary interfaces to enforce authorization and authentication policies on the service.

DL Management interacts with *Service Deployer* subsystem on the DHNs to load/configure/activate services and components on the node.

DL Management has to execute CSs and therefore it interacts with the CS Management service.

Other technologies

Java WS Core and related tools.

6.5.2.2 Package Repository

The Package Repository is the place where packages are stored. In order to be accepted a package must be prepared to be compliant with the Package Model that we have introduced in Sections 3.1 and 6.1.2. The "preparation" of a package must support any of its usage phases: DHN environment preparation, installation, customization, activation, update, replacement or removal. It is the responsibility of the package owner that registers the package as a DILIGENT Resource in the Dynamic VO Support Service to prepare the packages in such way. What can be prepared is an *installable unit package* (or IUP), a logical component that can be selected for installation. Physically, an installable unit package contains files to be installed, files that implement management operations (installation scripts), a set of manifest files which include a deployment descriptor that describes the install characteristics of the installable unit, and a descriptor that describes the binding (or physical locations) of the files in the unit.

The uploading phase is managed by the RegistrationUI portlet and the RegistrationService designed within the DVOS (see Section 7.5). These component allow to upload a new package into the Package Repository that validates the package (e.g. its conformity to the Package Model) and returns a positive/negative response. If the validation is ok, the Registration Service is in charge of registering the new resource in the DILIGENT VO and related sub-VOs while the Package Repository is responsible for the registration in the DIS for future discovering operations. When a DL Management component needs to retrieve a package, it first performs a discovery operation on the DIS-Registry and then it accesses the resource in the Repository.

Regarding the physical structure of a installable unit package, the Installable Unit Package Format Specification Version 1.0[13] will be investigated in the next stages of the design to understand if it is applicable as a formalism to model the logical Package Model presented.

Of course, not all DILIGENT resources are uploaded into the Package Repository, just the ones used to deploy and maintain a DL or to manage a new VO, including:

- Deployable services (WSRF service),
- Self-contained procedures (GridJobs),
- Shared/Stub libraries,
- Portlets.

Due to its scope, the component is not DL-specific, so it can be used by multiple DL Management components of different DLs at the same time.

The following figure depicts a typical DHN that hosts a Package Repository service.

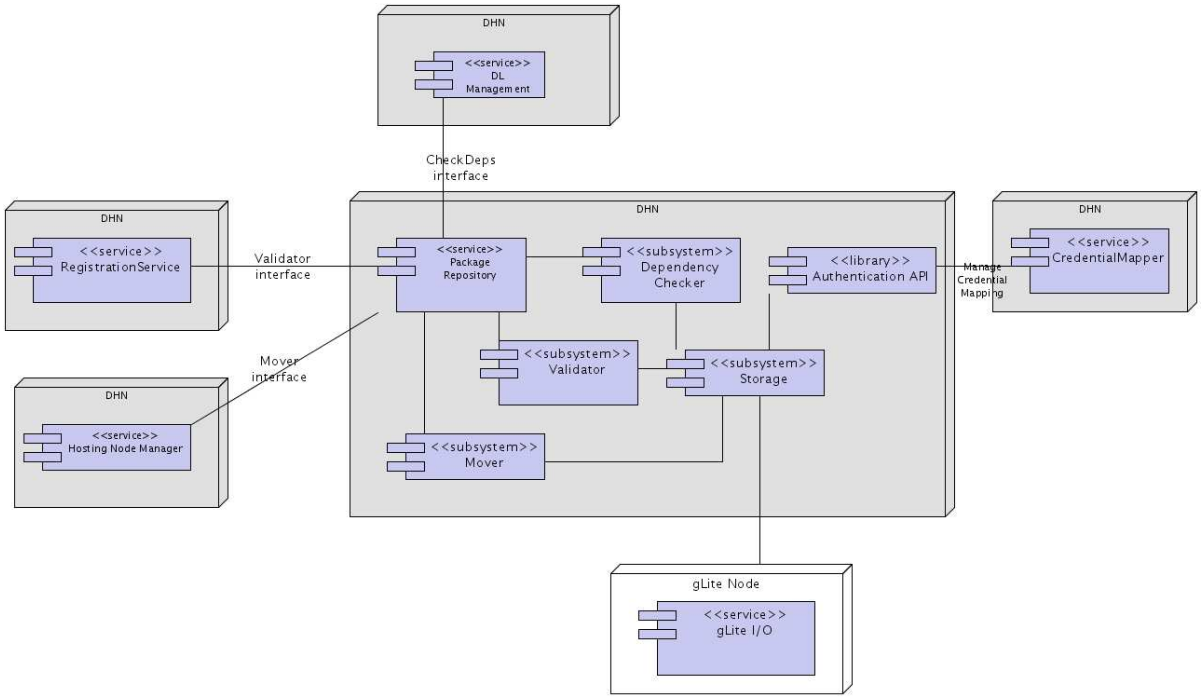

Figure 21. Package Repository Node - Deployment Diagram

Validator Subsystem

As suggested by its name, the role of the Validator is to validate the uploaded packages by verifying their conformity to the Package Model. Its interface is invoked by the RegistrationService when it receives a registration request of a new Resource labelled as Package.

Storage Subsystem

The Storage subsystem is the kernel of the Package Repository. The storage feature relies directly on the storage capabilities of the underlying Grid infrastructure. In particular, this subsystem interacts with the gLite I/O service and maintains a mapping table between

[13] http://www.w3.org/Submission/InstallableUnit-PF/

Logical File Names and the IDs assigned to the various software packages. To do this, it has to interact also with the CredentialMapper service in order to obtain valid gLite credentials.

In this way a Package Repository running instance can be moved on another DHN by just moving its mapping table.

Mover Subsystem

It is used to move the packages on the appropriate DHNs where they will be deployed. It relies on the GridFTP data transfer protocol.

Dependency Checker Subsystem

It builds and maintains structures that allow to discover dependencies relationships among the uploaded packages. It is typically queried by a DL Management service when it is elaborating the VDL definition criteria to compile the list of packages to be deployed to form a new DL.

6.5.2.2.1 State description

The Package Repository creates a single stateful resource that model the storage subsystem.

[to be detailed in D1.2.3]

6.5.2.2.2 Operations

- **Store**(input: package) – invoked by RegistrationService, it validates and stores a new software package
- **Delete** (input: package ID) – invoked by RegistrationService, it removes a package from the storage
- **CheckDependencies** (input: package ID) – invoked by DL Management, return the list of dependencies of a package
- **Retrieve** (input: package ID) – invoked by Hosting Node Manager, by using the Mover subsystem, it moves the package from the storage to the requester machine
- **RetrieveDescription**(input: package ID) – return a description of a package that contains a subset of the information included in the Package Model used to obtain a correct deployment of the package (typically this operation is invoked by DL Management service)
- **RetrieveDLMandatoryPackagesList**() – return the list of DLMandatory packages with their descriptions
- **RetrieveVOMandatoryPackagesList**() – return the list of VOMandatory packages with their descriptions
- **RetrieveDHNMandatoryPackagesList**() – return the list of DHNMandatory packages with their descriptions
- **ListRegisteredPackages**() – return the list of registered packages, filtered to contain only entries that the requester is enabled to use.

[to be detailed in D1.2.3]

6.5.2.2.3 Status description

The WS-ResourceProperties document associated to the single stateful resource provides a view of the list of the uploaded packages that allows to discover them and to ask for their retrieval.

[to be detailed in D1.2.3]

6.5.2.2.4 Dependencies and Requirements

DILIGENT services

The Package Repository uses the *Authentication API* (Section) developed within the DVOS Authentication area.

The Package Repository uses the *Authorization API* (Section 7.2) and *CredentialMapper* service (Section 7.1) developed within the DVOS Authorization area to implement the necessary interfaces to enforce authorization policies on the service.

It is expected that the RegistrationService (Section 7.5) invokes the Package Repository each time a new software package is uploaded.

gLite services

The Package Repository relies upon the data management services released by the EGEE project. It stores the packages in the SEs available in the underlying infrastructure by using the gLite I/O interface.

6.5.2.3 Hosting Node Manager

The third component of the Keeper Service is the Hosting Node Manager (HNM). It represents the minimal set of DILIGENT software that must be present on a DHN. As stated previously, a DHN is simply a node able to host DILIGENT services and related components. Since we rely on Java WS Core as the hosting environment, we can now reformulate this definition as the following:

A DHN is a node where a Java WS Core environment and a Hosting Node Manager are installed and which is configured to join the DILIGENT infrastructure.

Due to security concerns, this configuration must be created manually by the Resource Manager that makes available the node to the infrastructure.

Starting from a well-configured DHN, any DILIGENT software compliant with the Package Model can be dynamically deployed on the node. In particular, the first time that a DHN is up, any software labelled as "DHNMandatory" is automatically deployed on the DHN.

The DHN interacts directly with the Java WS Core software and related tools and modifies the package files to reflect the configuration requested by the DL Management.

The node management performed by the HNM involves the following tasks:

- Deploy/configure/undeploy new packages on the node
- Maintain and expose the node configuration to the deployed services
- Exchange data with the DL Management component
- Exchange data with the deployed services
- Push DHN and deployed service configurations to the local DIS-IP

To perform these tasks, the HNM has been designed as depicted in Figure 22.

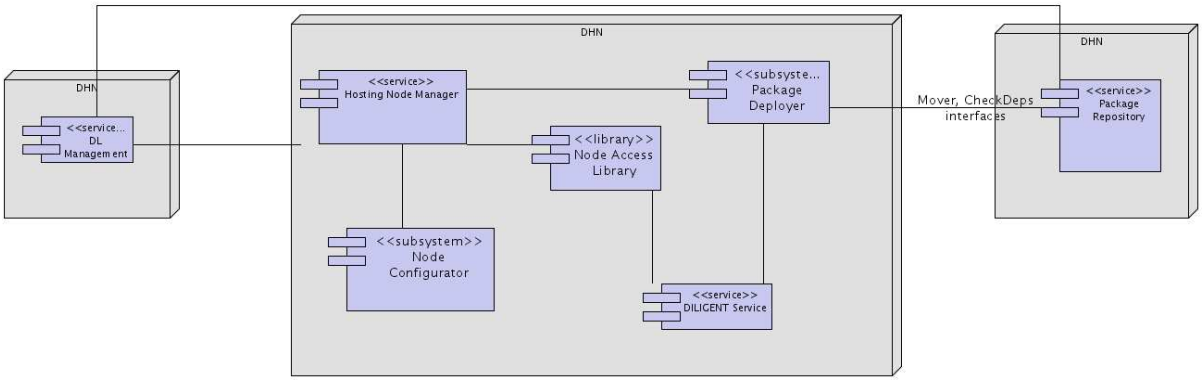

Figure 22. DHN Node - Deployment Diagram

Package Deployer subsystem

The deployment operations on a DHN are strictly related to the hosting environment. As stated previously, the selected hosting environment where we deploy our services is the Java WS Core developed by the Globus Alliance that it is able to host both WSRF and plain WS services.

Depending on the nature of the packages, four types of deployment exist:

- Deployment of a WSRF service
- Deployment of a library
- Deployment of a portlet
- Deployment of a Grid job

The deployment of a service is of course the most complex operation. To deploy a service into the Java WS Core container, a Grid Archive (GAR) file compliant with the GAR packaging format must be provided. This file contains all the files and information that the container needs to deploy a service. The following table reports the structure of a GAR file.

Directory/file	**Description**
Docs/	This directory contains service documentation files.
share/	This directory contains files that can be accessed or used by all services.
schema/	This directory contains service WSDL and schema files.
etc/	This directory contains service configuration files and a post-deploy.xml Ant script.
bin/	This directory contains service executables such as command line tools, GUI, etc.
lib/	This directory contains service and third party library files and any LICENSE files.
server-deploy.wsdd	This file is the server side deployment descriptor.
client-deploy.wsdd	This file is the client side deployment descriptor.

jndi-config-deploy.xml	This file is the JNDI configuration file.

In order to manage GAR archives, the build tool Ant[14] developed by the Apache Foundation comes with the Java WS Core. A number of available scripts provide a set of predefined Ant tasks (named *targets*) that help to perform common operations. A very low additional effort is necessary to customize the deployment tasks.

In order to avoid *ad hoc* deployment processes, in the next D1.2.3 a list of mandatory Ant targets will be reported.

The following is a set of typical actions performed by the Package Deployer when the DL Management contacts it to ask the deployment of a new package:

1. receive the package ID
2. contact the Package Repository to retrieve the list of requirements of that package and the list of other packages that must be deployed with it
3. verify that the list of requirements matches the actual node configuration
4. contact the Package Repository to retrieve the packages via a GridFTP connection
5. decompress each package file in a temporary folder
6. run the install scripts contained in the package to prepare the environment
7. deploy the GAR in the Java WS Core by running the appropriate ant task
8. modify the service installation files (WSDD, JNDI, Security Descriptor, etc.) to reflect the installation parameters
9. add the "profile" of the new service to the WS-ResourceProperties document of the HNM
10. push this information into the local DIS-IP
11. restart the container if it is necessary

From the Resource Model perspective, with the above steps the Package Deployer transforms one or more packages into a running instance of a service hosted in the DHN where the Package Deployer is running. The following picture is a subset of the Resource Model that shows the different classes involved in this stage.

[14] http://ant.apache.org/

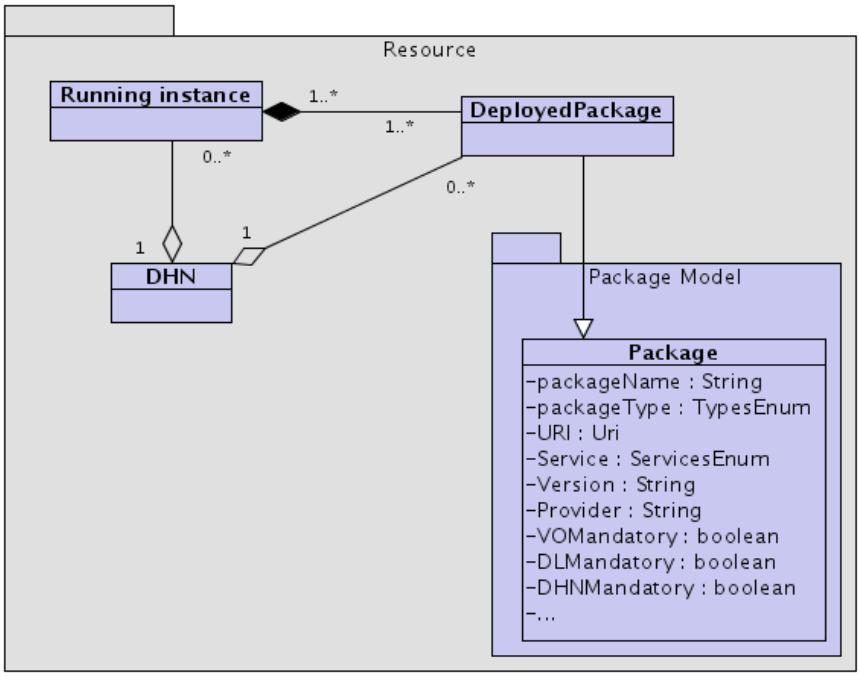

Figure 23. Package Deployer operations vs Resource Model

The "profile" of the new running instance (step 9) includes:

- a set of properties that "describes" the service implementation features (e.g. static properties) retrieved from a file included in the package (the format of this file will be provided in the D.1.2.3)
- a set of information retrieved from the running instance itself (via the UpdateServiceData() operation, see 6.5.2.3.2)
- a set of information added by the HNM about the DL, the creation time, the configuration parameter values, its URI and the original package.

This information allows the discovery of the running instance by other services. This kind of discovery is different from the one that is performed on the specific service data published by the service itself. In the former, the target of the discovery operation is to locate the running instance, in the latter the target is to find a particular WS-Resource of a running instance and obtain its EndPointReference in order to operate with it.

The step 10 (the restart of the container) implies that the resources created by the services must be persistent resources so they can survive container restarts. This is especially true for in-memory resources. In the WSRF implementation that comes with Java WS Core, a common way to reach this achievement is obtained when a resource implements the PersistentCallback interface.

In a similar (but simpler) way, the Library, Portlet and GridJob packages are deployed.

[to be detailed in D1.2.3]

Node Configurator subsystem

Each DHN has a set of basic configuration data. These data can be:

i. manually provided by the Resource Manager that makes available the node,
ii. passed at runtime by the DL Management Component or
iii. retrieved from the DIS (via the local DIS-HLSClient)

The manual configuration includes at least:

- the URL of the DIS-IC service to use
- the URL of the Package Repository to download packages from (it is important for downloading the DHNMandatory packages at first startup)
- the URL of the local AuthorizationService service (see Section 7.2)
- the installed software (third parties software)
- the hardware configuration

The Node Configurator is the module that supports the Resource Manager in the manual configuration and receives and manages the DL Management inputs.

[to be detailed in D1.2.3]

Node Access Library

This module is a shared library that exposes to the active services on the node a uniform API to query the current node configuration.

6.5.2.3.1 State description

The HNM service creates a single stateful resource that models the DHN that it is currently managing. The service does not need a factory service since a DHN can manage only one DHN per instance.

6.5.2.3.2 Operations

- **DeployPackage** (package ID, package configuration) – deploys the package on the DHN; then it configures the package according to the passed configuration. Figure 24 shows the set of operations performed after the invocation.

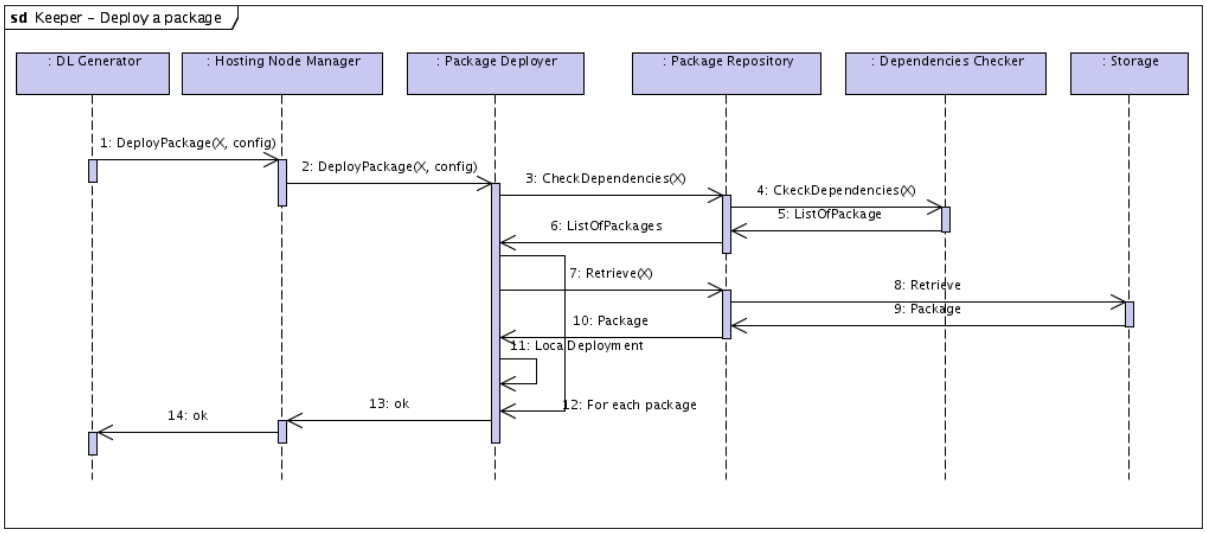

Figure 24. Add a Package to a DHN

- **UndeployPackage** (package ID) – undeploys the package passed as parameter
- **GetDeployedPackagesList**() – returns the list of deployed packages on the node
- **GetDHNConfiguration**() – returns a view of the DHN configuration
- **GetDHNStatus**() – returns a view of the current status of the DHN
- **DeployDHNPackages**() – deploys DHNMandatory packages

- **UpdateServiceData**(serviceID,serviceData) – updates the service instance data

[to be detailed in D1.2.3]

6.5.2.3.3 Status description

The WS-ResourceProperties document associated with the stateful resource that models the DHN offers a view on the DHN's state, including:

- the list of installed packaged
- the list of installed services and their configuration
- the DHN characteristics (conformed with the Host Model presented in Section 5.5.1.2)

6.5.2.3.4 Dependencies and Requirements

The HNM interacts, via the UpdateServiceData operation, with each local deployed service in order to retrieve information about it. This information is merged with the configuration parameters and other descriptive information (extracted from the package of the service) and published as WS-ResourceProperties of the HNM's WS-Resource.

These data are then used by other services to discover the running instances of services with particular configurations or behaviors. On the other hand, the specific application data of a service are published by the service itself via its own WS-ResourceProperties documents.

The HNM is an Information Provider for the local DIS-IP deployed in the DHN.

[to be detailed in D1.2.3]

6.5.2.4 K-UI

K-UI is a portlet hosted in the DILIGENT Portal that allows the DL Manager to interact with the DL Management service in order to manage its DLs.

6.5.2.4.1 Operations

- **CreateDL()**
- **DisposeDL()**
- **UpdateDL()**
- **ViewDLConfiguration()**
- **ViewDLHistory()**

[to be detailed in D1.2.3]

6.5.3 Deployment scenario(s)

[to be provided in D.1.2.3]

7 DYNAMIC VO SUPPORT SERVICE

The DILIGENT Service Oriented Architecture, based on the Web Services model, allows for an interaction among distributed and highly dynamic sets of services and resources. This distributed environment requires advanced authentication and authorization models to satisfy security constraints as reported in the DILIGENT functional specification [1]. The Dynamic VO Support (DVOS) service area is in charge to supply other DILIGENT services with a robust and flexible security framework as well as to provide services in order to manage VO concepts introduced in Section 3.7 of the test-bed functional specification. Moreover, DVOS enable DILIGENT services to join existing gLite-based infrastructures managing identity mapping among DILIGENT and gLite domains.

The DVOS area covers functionalities related to Resource Management (Section 4.3 of D1.1.1), to VO Management (Section 4.4 of D1.1.1), to Users and Group Management (Section 4.5 of D1.1.1) and to Notification Management (Section 4.6 of D1.1.1). Other non-functional requirements covered by DVOS are Authentication and Authorization issues as described in the D.1.1.1 Test-bed Functional Specification. All the functionalities related to DVOS are listed below.

Area: Authentication Management (Non functional)

- Certificates Issue and Revocation
- Certificate Validation
- Credential Storage
- Credential Retrieval
- Credential Renewal
- Credential Mapping

Area: Authorization Management

- Manage a VO (4.4.1)
- Create a VO (4.4.2)
- Add a Resource to a VO (4.4.3)
- Edit Resource Policy (4.4.4)
- Store Resource Policy (4.4.5)
- Add a User to a VO (4.4.6)
- Edit VO Roles (4.4.7)
- Store VO Roles (4.4.8)
- Edit User-Role Associations (4.4.9)
- Store User-Role Associations (4.4.10)
- Edit a VO (4.4.11)
- Remove a VO (4.4.12)
- Remove a Resource from a VO (4.4.13)
- Remove a User from a VO (4.4.14)
- List VOs (4.4.15)
- List VO Users (4.4.16)

- List User's VO-Resources (4.4.17)
- Get User's VO-Resources (4.4.18)

Area: Notification Management

- Notify (4.6.1)
- Notify Role (4.6.2)
- Notify User (4.6.3)
- Notify Group (4.6.4)

Area: User and Group Management

- Create a Group (4.5.1)
- Edit Group Profile (4.5.2)
- Store Group Profile (4.5.3)
- Add a User to a Group (4.5.4)
- Remove a User from a Group (4.5.5)
- Remove a Group (4.5.6)
- Add a User to DILIGENT (4.5.7)
- Edit User Profile (4.5.8)
- Request User Rights (4.5.9)
- Store User Profile (4.5.10)
- Remove User Profile (4.5.11)
- Remove a User from DILIGENT (4.5.12)
- Select Groups (4.5.13)
- Search for Groups by Details (4.5.14)
- Browse Groups (4.5.15)
- Select Users (4.5.16)
- Search for Users by Details (4.5.17)
- Browse Users (4.5.18)
- Invite a User (4.5.19)
- Propose User Rights (4.5.20)
- Invite a User to a DL (4.5.21)
- Invite a User to a Group (4.5.22)
- Invite a User to a Complex Object (4.5.23)
- Invite a Group (4.5.24)
- Propose Group Rights (4.5.25)
- Invite a Group to a DL (4.5.26)
- Invite a Group to a Complex Object (4.5.27)

Area: Resource Management

- Add a Resource to DILIGENT (4.3.1)
- Register a Resource (4.3.2)
- Edit Sharing Rules (4.3.3)

- Remove a Resource (4.3.8)

Due to the heterogeneity of the DVOS functionalities, this service is decomposed into five sub-packages. The diagram in Figure 25 summarizes this decomposition showing internal dependencies among packages. The internal structure of each functional package is explained in the following sections.

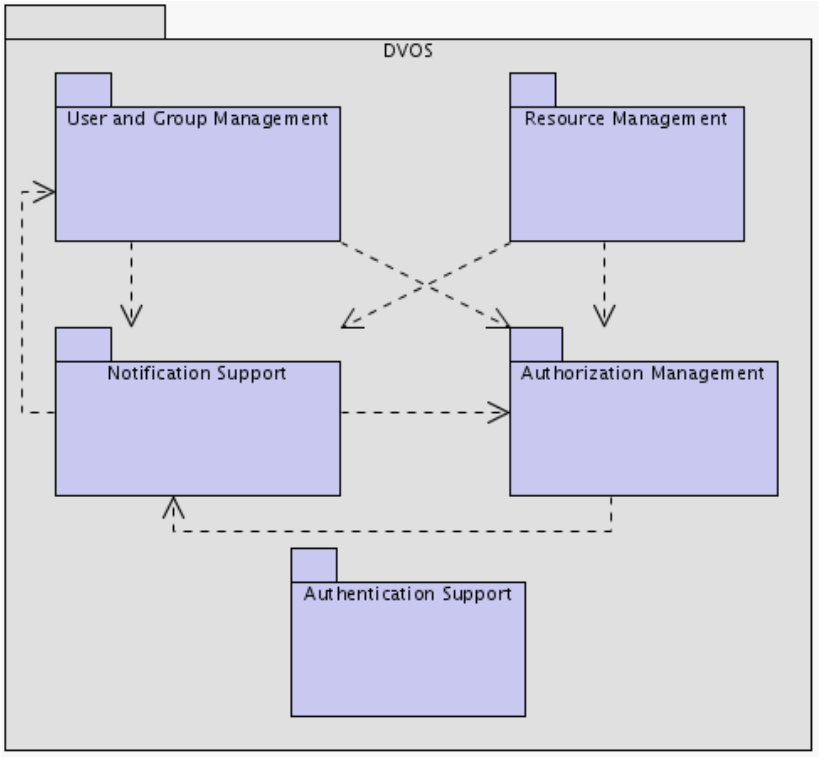

Figure 25. DVOS packages

In order to explain how Virtual Organizations are structured the VO Data Model is reported in Figure 26. *VirtualOrganization* is the main class; it includes references to DILIGENT users and resources. *Users* are identified through their Distinguished Name (DN) and *Resources* are identified through their UniqueID assigned to them at registration time (see Sections 7.4 and 7.5). A very important thing to point out is that Virtual Organizations do not define users and resources. Each VO includes references to existing users and resources only, thus they can be part of multiple Virtual Organizations at the same time. *Roles*, *Sharing Rules* and *Permissions*, on the contrary, are defined in VOs and inherited in Sub-VOs.

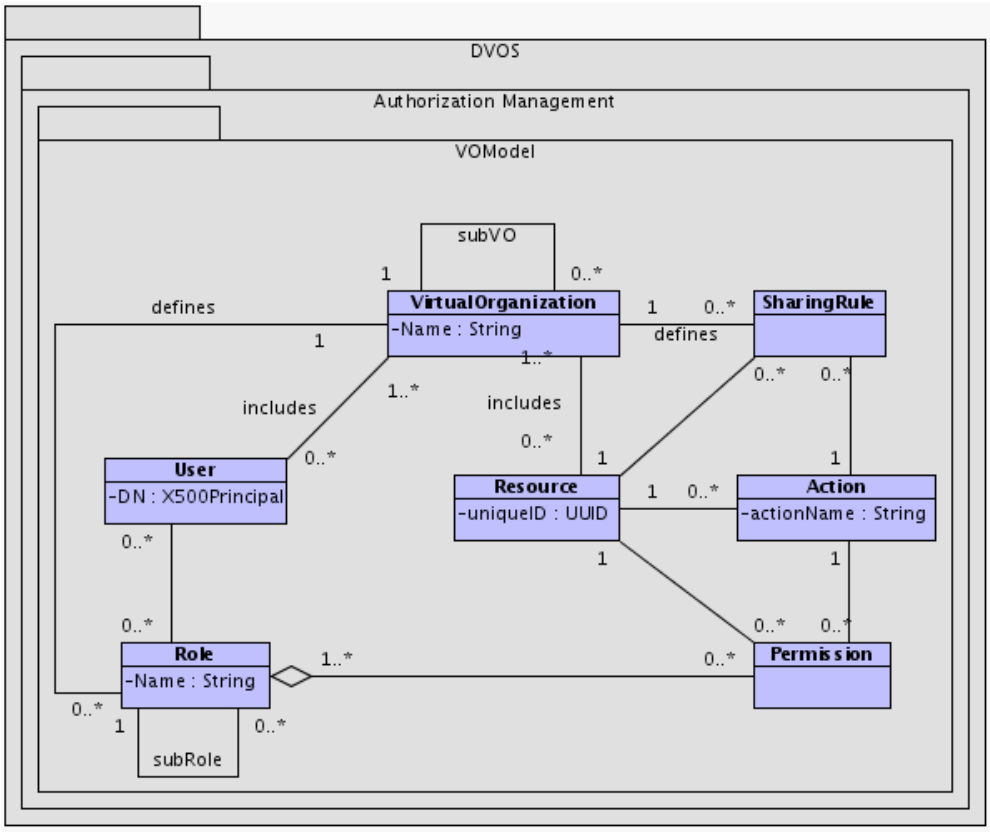

Figure 26. VO Data Model.

7.1 Authentication Support

Authentication support deals with user identity and credentials management issues. DILIGENT Authentication model is based on Grid Security Infrastructure (GSI)[15]. This implies the adoption of a Public Key Infrastructure (PKI) and the usage of X.509 End Entity Certificates (EEC) [16] released by a Certification Authority (CA) in order to authenticate DILIGENT users. GSI also supports delegation and single sign on in a distributed environment exploiting X.509 Proxy Certificates (PC).

In the DILIGENT infrastructure EECs can be assigned to the following DILIGENT entities:

- Users
- VOs
- Web Service instances (static of dynamic assignment)
- Service Container instances (static assignment)

The static assignment of EECs to Service Containers implies that a copy of credentials used by these entities must be available on the machine hosting the Container or the Web Service instance. The dynamic assignment is available for dynamically deployed services. A new X.509 certificate is generated at the end of the deployment process and provided to the service during its initialization. This certificate can then be used by the service to authenticate itself to the users.

[15] For a brief overview of GSI concepts and mechanisms see http://www.globus.org/toolkit/docs/4.0/security/key/index.html. An essential glossary about security terms is also available at http://www-unix.globus.org/toolkit/docs/3.2/gsi/key/glossary.html.

The main functionalities provided by this area are listed below. A brief description of these functionalities is also provided.

- *End Entity Certificate Issue*: issue of a new EEC and private key representing a DILIGENT identity. A trusted CA provides this functionality. This is the first step of the user registration process.
- *End Entity Certificate Revocation*: functionality provided by a trusted CA used to revoke a previously issued DILIGENT identity.
- *Certificate Validation*: functionality used by DILIGENT Services in order to verify the validity of the Proxy Certificate attached to a service request.
- *Credential Storage*: functionality periodically used by DILIGENT Users to create a middle-term copy of their EEC credentials[16]. After the creation, the copy will be available in the DILIGENT infrastructure. This functionality must be invoked from a machine where the EEC and the original private key of the user are available.
- *Credential Retrieval*: functionality used by DILIGENT Services (particularly the DILIGENT portal) in order to retrieve a short-term copy of the DILIGENT User credentials[17]. Such a copy will be generated from the middle-term copy of credentials.
- *Credential Renewal*: functionality used by gLite services running long time job in order to extend the validity of user credentials.
- *Credential Mapping*: functionality performed by DILIGENT services in order to obtain a valid gLite credentials from a DILIGENT one (see below).

Credential Mapping is a main issue in DILIGENT authentication; the need for this functionality depends on Certification Authorities acknowledged by DILIGENT platform. If these Certification Authorities are also acknowledged by external[18] gLite infrastructures accessible by the DILIGENT platform, then credential mapping can be avoided. Otherwise, the mapping is always required in order to access those external gLite infrastructures. In this case, the entire DILIGENT VO could share the same gLite account in order to access an existing gLite infrastructure. Moreover, the mapping does not have to be static and can be modified through new agreements between DILIGENT communities and gLite resources owners.

7.1.1 Use-Case View

An in-depth analysis of above functionalities and considerations leads to the Use-Case view reported in Figure 27. This view does not include functionalities related to the obtaining of the EEC from the CA. In order to request DILIGENT registration the user must have a valid EEC released by a trusted CA. Users can obtain such a certificate by contacting the DILIGENT SimpleCA (not shown in Figure 27).

DHNs and DILIGENT Services also need a valid certificate in order to be authenticated within the infrastructure. EECs for DILIGENT Hosting Nodes are obtained similarly to User EECs contacting a trusted CA. These certificates are stored in the node and used by the Hosting Node Manager only.

For EEC of dynamically deployed DILIGENT Services the use of an "official" CA does not constitute a suitable solution. At the time being, the only way to dynamically obtain such a certificate is to use an "ad hoc" Certification Authority. This CA should be configured to

[16] The user can set lifetime of the middle-term copy, it typically spans from a week to some months.
[17] The user can set lifetime of the short-term copy, typically a few hours.
[18] The "external" attribute is used to identify gLite infrastructures authorizing VOs different from the DILIGENT one (e.g.: EGEE).

accept certificate requests sent by DHN identities only. This way, only a DILIGENT Hosting Node can create a certificate for a DILIGENT Service.

Actors involved in authentication scenario are:

- *DILIGENT Users*, responsible to store a middle-term copy of their credentials in the repository.
- *DILIGENT VO Manager*, responsible to manage credential-mapping criteria.
- *Generic Services* (not necessarily Web Services) that can perform the *Renew Credential* operation (e.g. gLite services running long time jobs).
- *DILIGENT Services* during certificate validation and credentials mapping. Moreover, DILIGENT services can retrieve credentials through the *Get Credentials* use case. Particularly this functionality is used by DILIGENT portal during User login process to retrieve a short-term copy of user credentials.
- A brief description of the packages shown in Figure 27 is provided below.

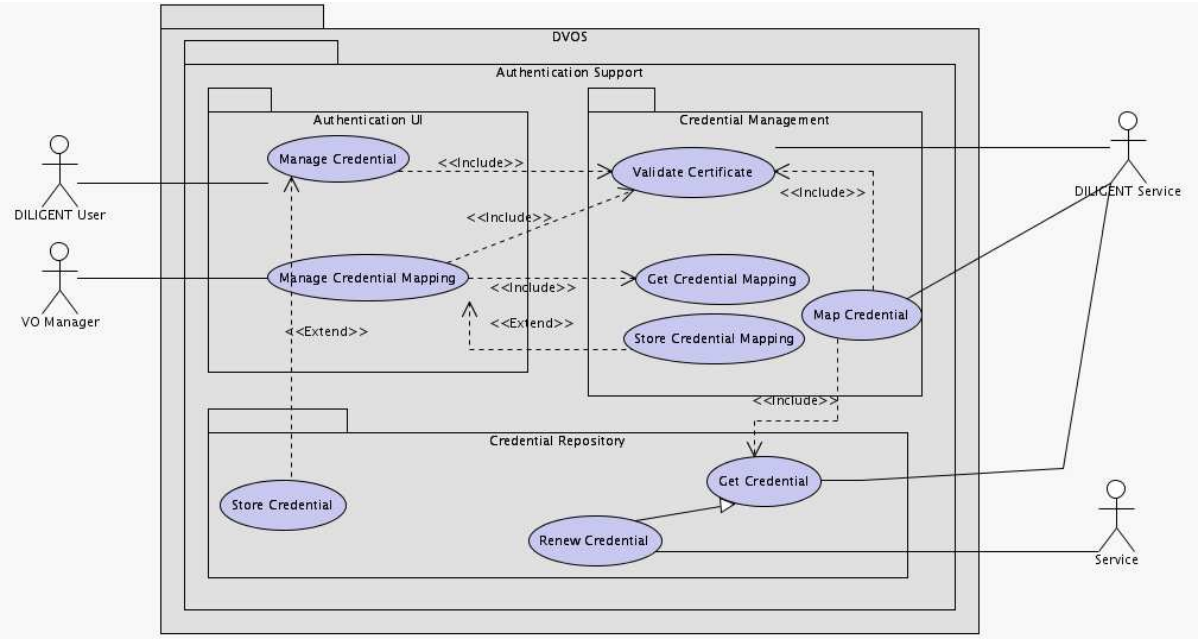

Figure 27. Authentication Support - Use-Case View

Credential Management package

This package is the core of the Authentication support. It contains functionalities that enable DILIGENT Services to validate the certificate of the caller as well as functionalities to map credentials in order to access gLite-based infrastructures outside the DILIGENT one. This package also contains functionalities to configure credential mapping.

- **Validate Certificate** - DILIGENT Services uses this functionality to validate the certificate of the caller. This operation is automatically performed by Java WS Core according to Authentication Methods defined for each service in the Service Security Descriptor file[19].
- **Map Credential** - DILIGENT Services use this functionality to retrieve a certificate to access gLite services. The need for this functionality depends on the gLite infrastructure being accessed. If that infrastructure does not trust DILIGENT Certification Authorities, an identity mapping is required. This use case must be included in each use case that implies access to gLite services.

[19] For a brief introduction about Service Security Descriptors see http://www.globus.org/toolkit/docs/4.0/security/authzframe/security_descriptor.html.

- **Get Credential Mapping** – This functionality allows a *DILIGENT VO Manager* to retrieve a previously stored credential mapping.
- **Store Credential Mapping** - This functionality allows a *DILIGENT VO Manager* to store a credential mapping.

Authentication UI package

Through the Authentication UI package *DILIGENT Users* can store their own credentials in the repository and a *DILIGENT VO Manager* is allowed to manage credential-mapping criteria. These criteria are used during the *Map Credential* operation to enable access to existing gLite-based Grid infrastructures.

- **Manage Credential** - Through this use case, *DILIGENT Users* are enabled to create a middle-term copy of their End Entity Certificate and store it in the Credential Repository. This copy can be subsequently used to obtain short-term copies of user's certificate to use during user sessions.
- **Manage Credential Mapping** – This use case enables *DILIGENT VO Manager* to modify credential mapping used to access existing gLite infrastructures. For each gLite infrastructure, outside the DILIGENT one, an alternative credential can be set.

Credential Repository package

This package contains functionalities related to the credential repository. Use cases of this package enable DILIGENT services to retrieve user credential and generic services (DILIGENT and gLite) to renew it during execution.

- **Store Credential** - This operation allows users to store a middle-term copy of their DILIGENT or gLite certificates in the repository. Each user set the lifetime of the stored copy as well as the password needed to retrieve the short-term copy of credentials.
- **Get Credential** - In order to get a short-term copy of user credentials the username and password previously set by the user are needed.
- **Renew Credential** - In order to renew short-term user credentials a valid copy of credential to renew is needed. Only DILIGENT hosts can perform this use case.

7.1.2 Logical View

Main entities involved in certificate management are:

- One or more trusted CAs, which are in charge to perform *End Entity Certificate Issue* and *End Entity Certificate Revocation*.
- A Credential Repository (CR) which is in charge to provide *Store Credential* and *Get Credential* functionalities as well as the *Renew Credential* operation.
- A Mapping Service, which is in charge to *Store Credential Mapping* and to perform the *Map Credential* operation.
- A set of *Authentication APIs* allowing clients to use functionalities provided by this package.
- A portlet enabling *DILIGENT VO Manager* to configure credential mapping.

Users do not always login to the DILIGENT infrastructure from the same computer. The credential repository is needed in order to allow users and services to retrieve their credentials whenever and wherever they need it, without worrying about managing private key and certificate files.

The analysis of the use cases in Figure 27 leads to the following logical view. Figure 28 contains the main classes belonging to the Authentication Support package.

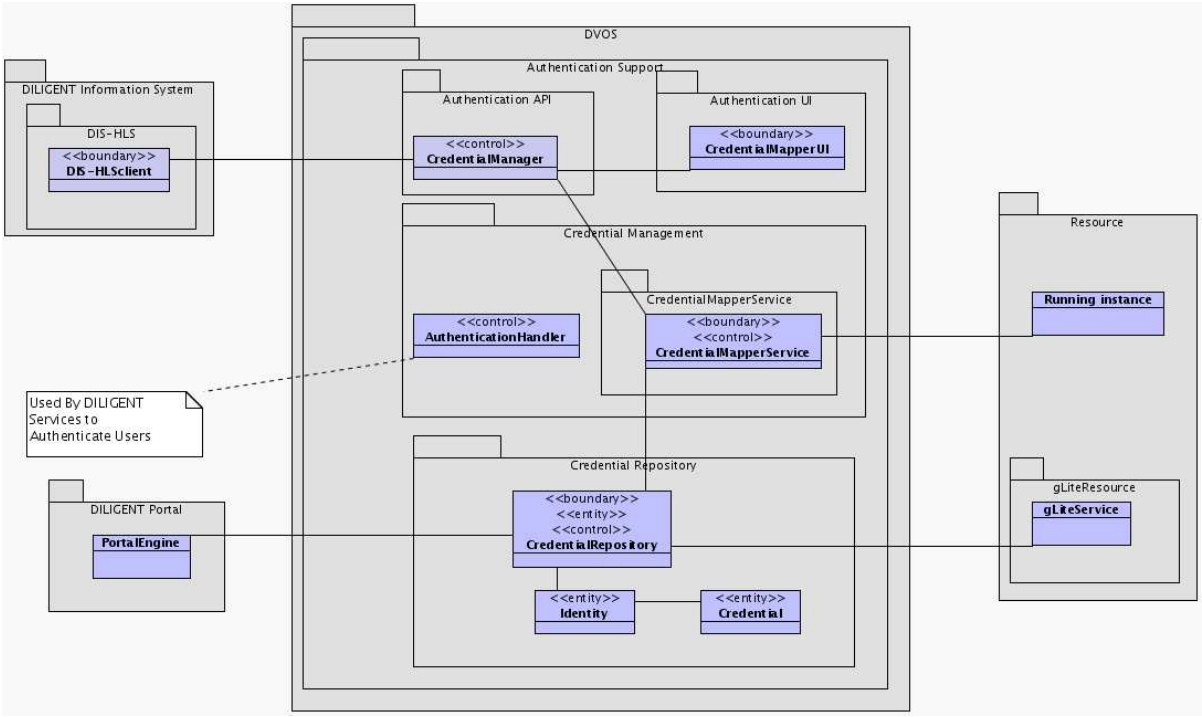

Figure 28. Authentication Support - Logical View

AuthenticationUI package

This package contains the User Interface part of the Authentication Support package.

- **CredentialMapperUI** – This class provides a *DILIGENT VO Manager* with access to the *CredentialMapperService*. This service can be used to configure alternative credentials to use in order to access external gLite infrastructures.

Authentication API package

Classes of this package provide DILIGENT Resources with a set of APIs to use DILIGENT Authentication functionalities. The aim of these classes is to hide details of the Authentication Support package to clients, allowing them to decouple their behaviour from the internal structure of Authentication Support.

- **CredentialManager** – This class allows clients to manage credentials associated to users and services. It interacts with the DIS-HLSClient in order to discover *CredentialRepository* and *CredentialMapperService* locations.

Credential Management package

- **CredentialMapperService** - This component maintains credential mapping between DILIGENT credential and those used to access external gLite middleware.
- **AuthenticationHandler** - This handler class allows DILIGENT services to authenticate incoming requests through the validation of the Proxy Certificates attached to service requests. This class is provided and managed by the underlying GSI implementation. Depending on the security configuration of the service, the Java WS Core automatically performs the Proxy Certificate validation on incoming requests.

Credential Repository package

This package contains classes related to Credential Repository. Credential Repositories store middle-term copies of credentials of DILIGENT users and services as well as credentials used to access external gLite middleware.

- **CredentialRepository** - This class models the repository of credentials. It provides *DILIGENT Users* with the *Store Credential* functionality as well as other services (DILIGENT and gLite) with the *Get Credential* and *Renew Credential* functionalities.
- **Identity** - This entity class models the *DILIGENT User* account in the credential repository.
- **Credential** - This class models a single set of credentials associated with a *DILIGENT User*.

7.1.3 Deployment View

Figure 29 shows a deployment view of services provided by the Authentication support package.

The Authentication Support package is decomposed in the following components:

- **CredentialMapperUI** – This component is a portlet that provide *DILIGENT VO Manager* with a graphical view of authentication functionalities provided by the Authentication API library.
- **AuthenticationAPI** – This library contains classes allowing DILIGENT Resources to invoke authentication functionalities.
- **CredentialMapper** – This is a WSRF Service deployed on a DILIGENT Hosting Node. Its role is to maintain the mapping between DILIGENT credentials and other credentials used to access external gLite middleware.
- **CredentialRepository** - As shown by the stereotype in the diagram the *CredentialRepository* component is a non-DILIGENT component and must be instantiated in a non-DILIGENT Hosting Node. This requirement arises from the need of gLite services to perform the *Renew Credential* operation. *CredentialRepository* is also used by DILIGENT portal during *Login* functionality in order to retrieve user credentials.

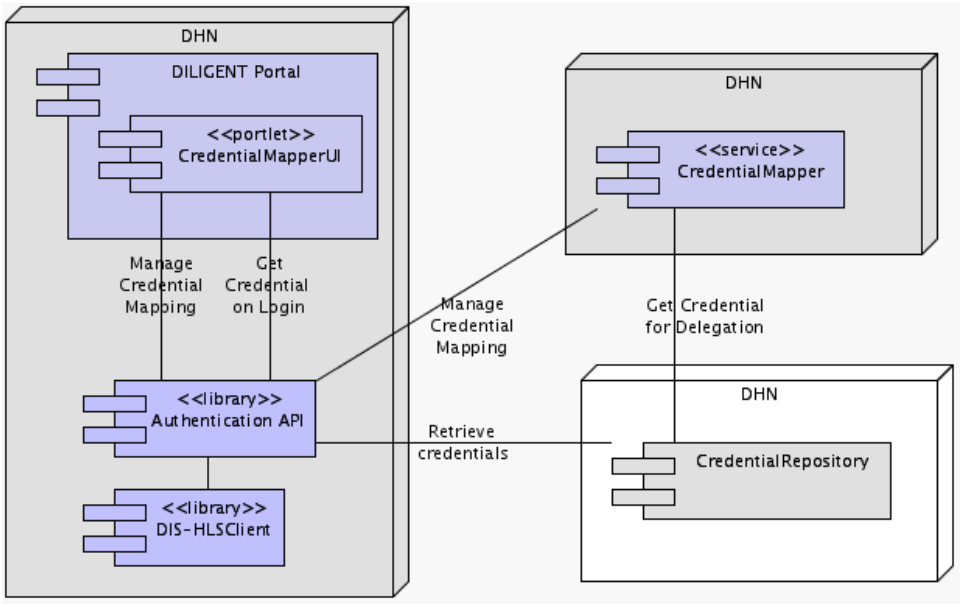

Figure 29. Authentication Support - Deployment View

7.1.4 Service design

7.1.4.1 Design Considerations

Credential Repository Considerations

Authentication issues in DILIGENT infrastructures mainly deal with certificate management. The need for DILIGENT users to access the infrastructure from different workstation and the dynamic deployment of DILIGENT services lead to the adoption of a Credential Repository. Such a repository allows DILIGENT portals to retrieve short-term copy of DILIGENT user's certificates. It also enables DILIGENT services to obtain a local copy of these certificates from the repository.

Myproxy [21] is a credential repository that support functionalities identified in the Section 7.1 of this document. Its use as DILIGENT credential repository comes from many reasons: its stability, due to its wide adoption in Grid environments; its security features, passwords to decrypt private keys are not stored in the repository; and the need of gLite services to perform credential renewal.

Credential Mapping Considerations

Another issue in Authentication Support design is the identity mapping. It is required in order to access external gLite infrastructures that do not trust Certification Authorities acknowledged by DILIGENT. To access these infrastructures DILIGENT credentials must be converted to external credentials. The Credential Mapping service aims to address this problem allowing *DILIGENT VO Manager* to set alternative credentials to use.

Service Credential Considerations

In the DILIGENT environment, services can be dynamically deployed on different DILIGENT Hosting Nodes, thus enabled to create and obtain their own credentials automatically. These credentials are needed to perform service authentication to the clients. For the time being Certification Authorities only allow to create service credentials through procedures that involve human intervention. In DILIGENT infrastructure the needs to automatically create credentials is solved through the adoption of a DILIGENT Certification Authority[20]. This Certification Authority will be in charge to generate credentials for services deployed on a node without human intervention. The container hosting the new service instance requests these credentials as part of the service deployment operation. Created credentials are stored (encrypted) in the node hosting the service and provided to the service by the container. The public certificate of the service credential is returned to the Keeper at the end of the deployment operation in order to allow it to set authorization for the service that was just deployed. The container itself is in charge to manage credentials available to services running on a node in order to avoid the abuse of these credentials.

7.1.4.2 Components

7.1.4.2.1 CredentialMapper

7.1.4.2.1.1 State description

[20] The alternative to this approach is to disable authentication of services deployed dynamically, but this limitation seems to be too restrictive for the DILIGENT infrastructure.

The pattern adopted for the *CredentialMapper* design is the WS-Resource Singleton Pattern[21]. The internal structure of the WSRF service is shown in Figure 30 Only one *CredentialMapperResource* is created by the *CredentialMapperHome* at service start-up and used to store the identity mapping between alternative credentials.

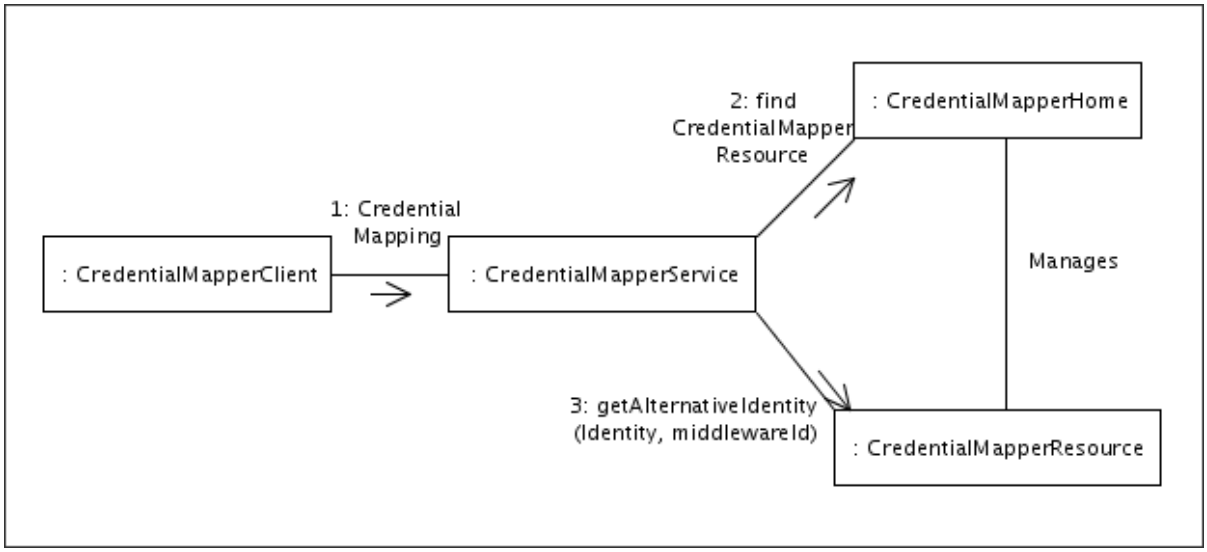

Figure 30. Authentication Support - CredentialMapper structure

7.1.4.2.1.2 Operations

- **mapCredential**(middlewareId)::GSSCredential – This operation allows DILIGENT services to retrieve alternative credentials to access an external gLite infrastructure. Each of these infrastructures (e.g. EGEE) has a different name. The *middlewareId* parameter is the identifier of the external gLite infrastructure. The choice of the infrastructure to access it is up to the DILIGENT service invoking this operation. Figure 31 shows the behaviour of the *mapCredential(…)* operation. Alternative credentials must be stored in the repository and can be retrieved only by the *CredentialMapperService*. This allows gLite services to perform credential renewal (steps 10-11). The *GSSCredential* object returned by this operation can be used to authenticate DILIGENT service accessing the gLite external middleware.

[21] For a brief overview of main design patterns for WSRF Services see http://gdp.globus.org/gt4-tutorial/multiplehtml/pt02.html (chapters 4 and 5).

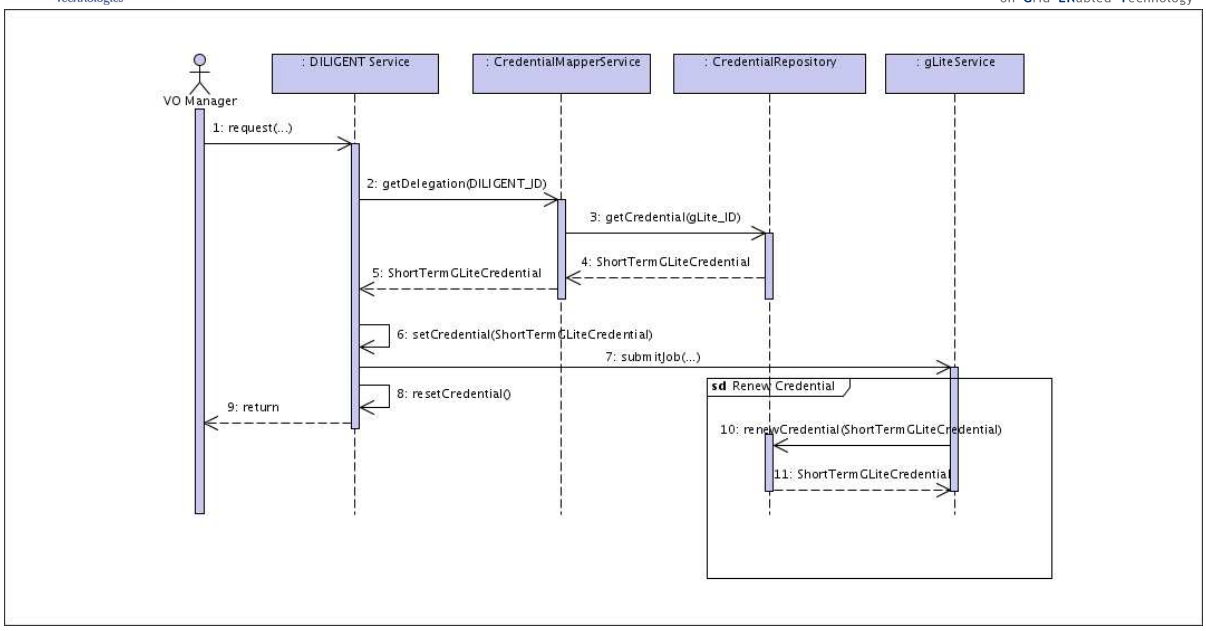

Figure 31. Authentication Support - Map Credential sequence diagram

The DILIGENT service contacts the *CredentialMapper* service providing the certificate of the DILIGENT User in order to obtain the alternative credentials (step 2). These credentials are obtained from the *CredentialRepository* (step 3 and 4) and forwarded to the DILIGENT Service (step 5). When received these credentials are set in the client used to access the gLite service (step 6) enabling access to the gLite service, e.g. to submit a job (step 7). After job submission, credentials are reset to the original ones (step 8).

- **setMapping**(middlewareId, alternativeIdentity, scope) – This operation allows *DILIGENT VO Manager* to set the alternative credentials to use to access external gLite infrastructures. The *middlewareId* parameter is the identifier of the external gLite infrastructure. The *alternativeIdentity* parameter is the identifier of the account (on the credential repository) from which to retrieve alternative credential. The *scope* parameter allows specifying the set of DILIGENT users entitled to use these alternative credentials (the default is set to all DILIGENT users).
- **resetMapping**(middlewareId, alternativeIdentity) – This operation removes from the CredentialMapper the association between the *middlewareId* and the *alternativeIdentity* specified as parameters. Only a *DILIGENT VO Manager* can perform this operation.
- **getMapping**(middlewareId, alternativeIdentity) – This operation returns a list of alternative credentials stored in the CredentialMapper. Only a *DILIGENT VO Manager* can perform this operation.

7.1.4.2.1.3 Profile description

The *CredentialMapperService* profile registered in the DIS does not include any particular information outside the standard WSRF-service profile.

The security profile of the *CredentialMapperService* is configured as follows:

- **CredentialMapperServiceSecurityDescriptor1**: default
  - **Authorization Chain**: only VO-level authorization is performed through a handler using *Authorization API* provided by DVOS.
    - **Authorization Criteria**: All *DILIGENT User*s are entitled to perform the *mapCredential* operation. Only a *DILIGENT VO Manager* can

perform the *setMapping(…)*, *resetMapping(…)* and *getMapping(…)* operations.

- **Default Settings**
  - **Default Identity Scenario**: Service
  - **Default Authentication Methods**:
    - **GSISecureMessage** (Integrity & Privacy)

7.1.4.2.1.4 Status Description

The status of the CredentialMapper published in the DIS includes the followings information:

- **MiddlewareIdList:** the list of middleware accessible through alternative credentials stored in the CredentialMapper service.

[to be detailed in D1.2.3]

7.1.4.2.1.5 Dependencies & Requirements

This service relies upon the following entities:

DILIGENT services

- DVOS: CredentialRepository
- DIS: DIS-HLSClient

7.1.4.2.2 CredentialRepository

As previously described, the DILIGENT CredentialRepository will be implemented through a MyProxy server. As this is not a WSRF service, most paragraphs of this section are empty. This section does not aim to describe the behaviour and features of MyProxy, but to explain how this component is used within the DILIGENT infrastructure.

7.1.4.2.2.1 Operations

This section explains some MyProxy usage scenario in the DILIGENT infrastructure.

User registration and login - Figure 32 shows the store and retrieval of user's credentials. The DILIGENT User performs step 1 from a machine where his EEC and private key are available. The user sets the lifetime of the middle-term credentials. This operation is performed without the support of the DILIGENT portal. Steps 2 to 4 show the user login and the retrieval of short-term credentials performed by the DILIGENT portal. Users can perform this operation on every machine equipped with a browser supporting HTTPS features (in order to securely send username and password to the DILGIENT portal). Connection between the DILIGENT Portal and the CredentialRepository is protected and encrypted using the TLS security protocol (with mutual authentication). Short-term credentials of the user are stored in the portal and deleted at the end of his session. These credentials should not be directly transmitted to other DILIGENT services or to the client, but only used to authenticate user's actions and to perform credential delegation.

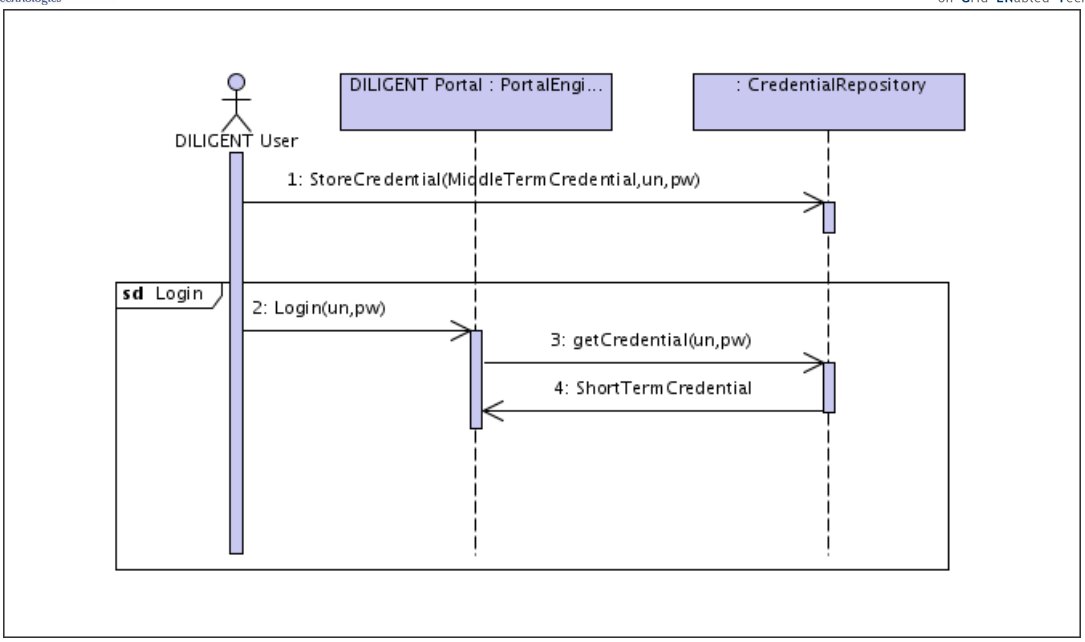

Figure 32. Authentication Support - Registration and Login sequence diagram

DILIGENT Service Credential retrieval – This usage scenario describes how to use MyProxy to retrieve credentials used by DILIGENT services.

[to be detailed in D.1.2.3]

7.1.4.2.2.2 Profile description

NONE (This service is not registered in the DIS)

7.1.4.2.2.2.1 Security Description

NONE (This service is not registered in the DIS)

7.1.4.2.2.3 Status Description

NONE (This service is not registered in the DIS)

7.1.4.2.2.4 Dependencies & Requirements

This service relies upon the following entities:

DILIGENT services

NONE

gLite services

NONE

Other technologies

- MyProxy server

7.1.4.2.3 Authentication APIs

7.1.4.2.3.1 Description

These APIs are composed by two separate subsets of classes. The former set can be used to manage service credentials (not stored in the repository) and to map DILIGENT

credentials. The latter set is provided by the MyProxy support and can be used to contact the credential repository in order to store and retrieve user's credentials[22].

7.1.4.2.3.2 Usage

[to be provided in the D.1.2.3]

7.1.4.2.4 CredentialMapperUI

[to be provided in the D.1.2.3]

7.1.4.2.4.1 Operations

[to be provided in the D.1.2.3]

7.1.4.2.4.2 Interaction with other components

[to be provided in the D.1.2.3]

7.1.4.3 Deployment scenario(s)

[to be provided in the D.1.2.3]

7.2 Authorization Management

Multiple levels of authorization can be defined for DILIGENT Resources. This package provides support for VO-level Authorization. It originates from the VO model as defined in section 7. Service designers can also define and use other finer-grained levels of authorization.

The X.509 standard that define Proxy Certificates format allows a PC to carry authorization information as certificate extensions [16]. These extensions are included in the PC during its creation. This functionality allows authorization mechanisms at every node to locally resolve authorization issues, without the need to contact a remote authorization service. Nevertheless, this model binds the lifetime and the extent of authorizations to the information contained in the Proxy Certificate. In the DILIGENT infrastructure, where resources are dynamically created and destroyed, a more dynamic authorization model is needed. DILIGENT Resources must be enabled to grant (or deny) access based on up-to-date authorization information. Proxy Certificates of DILIGENT users will carry only the name of the VO where the user operates. The authority maintaining the current state of that VO must be asked in order to grant users permissions to access DILIGENT Resources. The decision on when to contact this authority during resource runtime is up to the resource implementation and thus up to the resource designer.

The DILIGENT authorization model relies on the underlying Java WS Core authorization framework. It enables DILIGENT Services to separate security issues from service specific behaviour. This is achieved using a chain of authorization handlers managed by the Service Container. These handlers are asked during evaluation of incoming requests in order to permit or deny access to a service. The authorization chain can be configured through an XML file named "Security Descriptor"[23]. Each DILIGENT Service is in charge to implement these authorization handlers in order to enforce VO-level and resource-specific authorization policies.

---

[22] Javadoc description of MyProxy APIs is available at http://www-unix.globus.org/cog/distribution/1.2/api/org/globus/myproxy/package-summary.html

[23] For a detailed description of Java WS Core security settings see http://www.globus.org/toolkit/docs/4.0/security/authzframe/security_descriptor.html.

7.2.1 Use-Case View

The analysis of the DILIGENT VO model (see [1], Section 3.7) points out the need for a structure capable of maintaining hierarchical authorization information. This information will be used by DILIGENT services during handling of incoming requests.

Actors related with the authorization management package are *VO Managers* and *DILIGENT Resources*. The former are in charge of inspect and edit VO hierarchy as well as to set authorization policies at VO-level. The latter are in charge to enforce authorization policies on incoming requests.

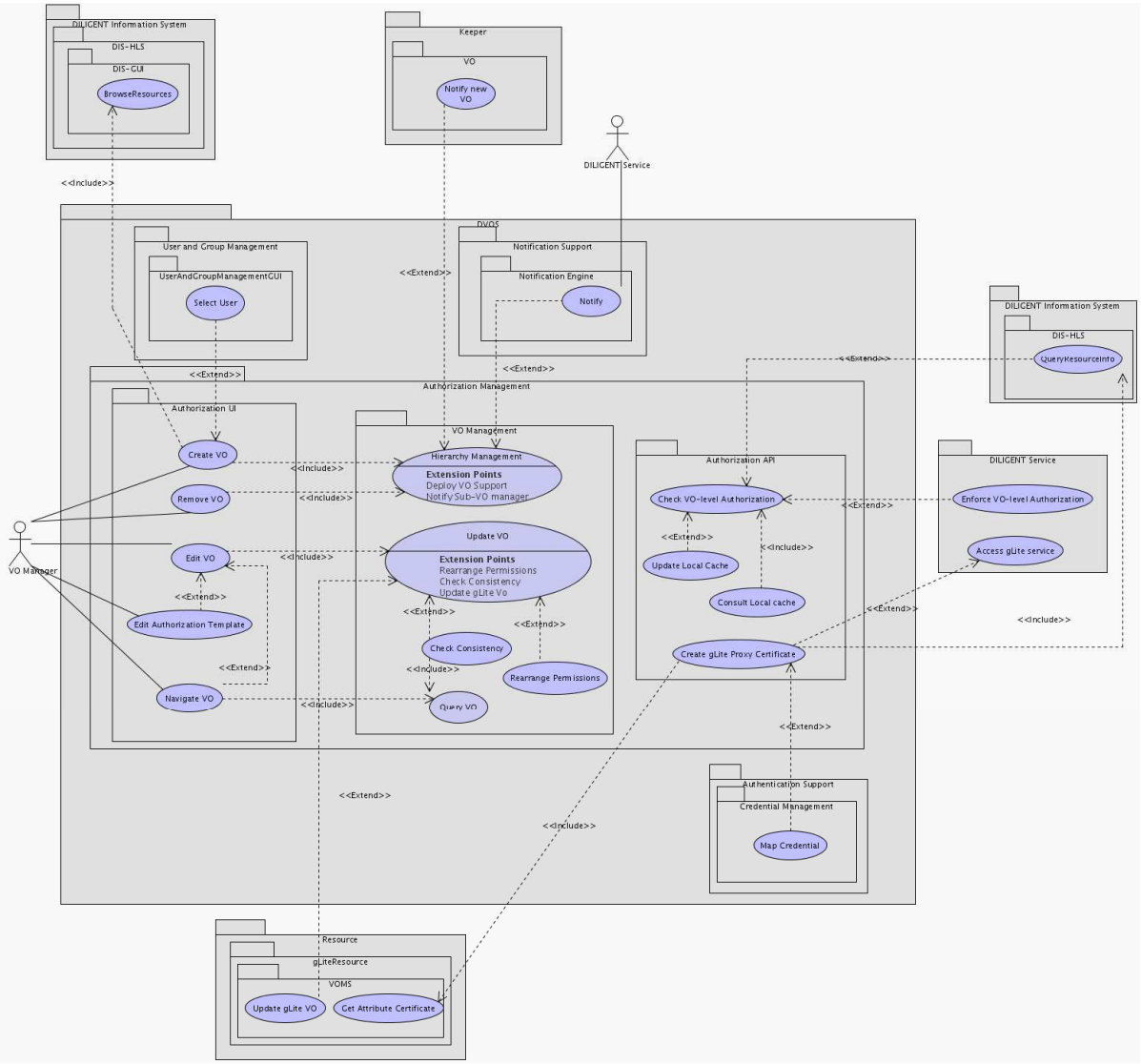

Figure 33. Authorization Management - Use-Case View

Authorization UI package

VO Managers interact with VO Management through the Authorization UI package. It includes the following high-level functionalities.

- **Create/Remove a VO** – This functionality enables *VO Managers* to create and remove sub-VOs. These use cases respectively cover the user interface part of functionalities named "Create a VO" and "Remove a VO".
- **Edit VO** - It includes a set of UI-related functionalities for the modification of the VO status. This use case covers the user interface part of functionalities named "Add a Resource to a VO", "Edit Resource Policy", "Store Resource Policy", "Add a User to a

VO", "Edit VO Roles", "Store VO Roles", "Edit User-Role Associations", "Store User-Role Associations", "Remove a Resource from a VO", "Remove a User from a VO".

- **Navigate VO** - It allows VO Managers to navigate through the VO Hierarchy and to inspect VO status. This use case covers the user interface part of functionalities named "List VOs", "List VO Users", "List User's VO-Resources", and "Get User's VO-Resources".
- **Edit Authorization Template** – This use case allows *VO Managers* to set Authorization policies to be applied when a new RunningInstance is included in their VO. When a new instance of a service is deployed, VO-level permissions must be set to enable it to interoperate with other DILIGENT running instances. The Keeper, as *VO Manager*, is in charge to perform these settings after the service deployment. *VO Managers* must define permissions to apply when a new instance of the service is deployed in their VOs. These permissions are defined accordingly to sharing rules set by the *Resource Managers* during the package registration (see the description of *Edit Sharing Rules Template* use case in Section 7.5.1).

VO Management package

This package is the core of the VO Authorization model; it contains functionalities to manage structure and behaviour of DILIGENT VO and VO hierarchy.

- **Check Consistency** - Any time permission over a given resource is granted to a role, sharing rules (defined by the resource manager on that resource) need to be respected. This use case models such consistency check, which guarantees that no role (and thus no user) can use a resource beyond the authorization granted to the whole VO by the *Resource Manager*.
- **Rearrange Permissions** - Similarly to the previous one, this functionality enforces consistency between resource sharing rules and permissions granted to roles within a VO. Any time a *Resource Manager* changes the way a resource is shared with a VO (i.e. by modifying the corresponding sharing rules), permissions need to be rearranged accordingly. In particular, any permission that does not fulfill any sharing rule is removed.
- **Authorization Decision** - This functionality enables a generic service to determine if a given user is allowed to perform an action on a particular resource. This is a core functionality of the Authorization Management package.
- **Query VO** - According to D1.1.1 Functional Specification, this high-level inspection functionality comprises a number of sub-functionalities for the investigation of the VO status.
- **Update VO** - According to D1.1.1 Functional Specification, this high-level management functionality comprises a number of sub-functionalities for the modification of the VO status. Among these functionalities, *SetRolePermissions* needs to be mentioned because it is used to grant resource usage permissions to DILIGENT users. The *Check Consistency* functionality is used any time a new permission is granted in order to respect sharing rules set by *Resource Manager*.
- **Hierarchy Management** - This use case models the core behavior of "Create a VO" and "Remove a VO" functionalities. The creation of a new VO involves the creation of resources needed for the management of the VO itself. In particular, the Keeper service is contacted to deploy new services needed by the new VO (see Section 6.2). Virtual Organizations in DILIGENT are organized in a tree-like structure. Removal of a VO in the structure implies removal of the whole sub-tree rooted at that VO. In order to allow the preservation of sub-VO resources, the removal of the VO sub-tree delayed and preceded by a notification to all the VO Managers of the sub-VOs.

Authorization API package

DILIGENT services need a set of functionalities in order to locally enforce the VO-level authorization policies. This package provides these functionalities and adds local caching support to speed up the authorization checking process. In a distributed system, where authorization information is spread over network nodes, the adoption of such a mechanism can greatly improve performance. The use of the cache is not mandatory for DILIGENT services, but is recommended for resources whose authorization information changes slowly.

- **Check VO-level Authorization** - This is the core functionality used by services in order to enforce VO-level authorization policies on resources. This functionality allows DILIGENT services to obtain an up-to-date authorization decision in order to grant users permission to access a given resource. Given the name of the VO where the user operates, this functionality return an authorization decision based on the triple <User, Resource, Action>. Depending on service configuration this functionality can use the local cache to retrieve needed information or directly contact the VO Management package.
- **Consult Local Cache** - In order to speed-up the authorization decision on the Resource side as well as to reduce network traffic, the resource can use locally cached authorization information. Mechanisms will be provided to guarantee that authorization information retrieved locally from the cache is up-to-date with respect to those maintained by the VO Management package.
- **Update Local Cache** - Local caches must be updated to avoid inconsistency with authorization information managed by the VO Management package. Each authorization decision provided by the VO Management package has an expiration time. Expired authorization information is automatically deleted from the cache, but new information for a given resource is reloaded only when a new request for that resource is received. Moreover, each DILIGENT Resource can set its own cache update frequency to further reduce cache latency.
- **Create gLite Proxy Certificate** – This functionality is used by DILIGENT Services to create a gLite enabled Proxy Certificate. Such a certificate, containing an Attributes Certificate provided by VOMS, is needed to access gLite Services. The Map Credential functionality of the Authentication Support is used to obtain a Proxy Certificate acknowledged by a gLite middleware infrastructure. This use case exploits the discovery functionality provided by the DIS in order to locate the VOMS instance to contact.

7.2.2 Logical View

Entities involved in providing functionalities described in paragraph 7.2.1 are:

- A federation of *Authorization Services* in charge to maintain VO Authorization information. Through their interactions, these services assure global consistency of the entire DILIGENT VO hierarchy. They supply functionalities belonging to the VO Management package.
- A set of *Authorization APIs* allowing DILIGENT Resources to use functionalities provided by *DVOS Authorization Management* without the need to know the internal architecture of this package. These APIs are also in charge to provide users with the authorization cache support.
- A User Interface allows *VO Managers* to use *Authorization APIs* functionalities. This interface models use cases of the Authorization UI package.

Main classes of the Authorization package are depicted in the following diagram.

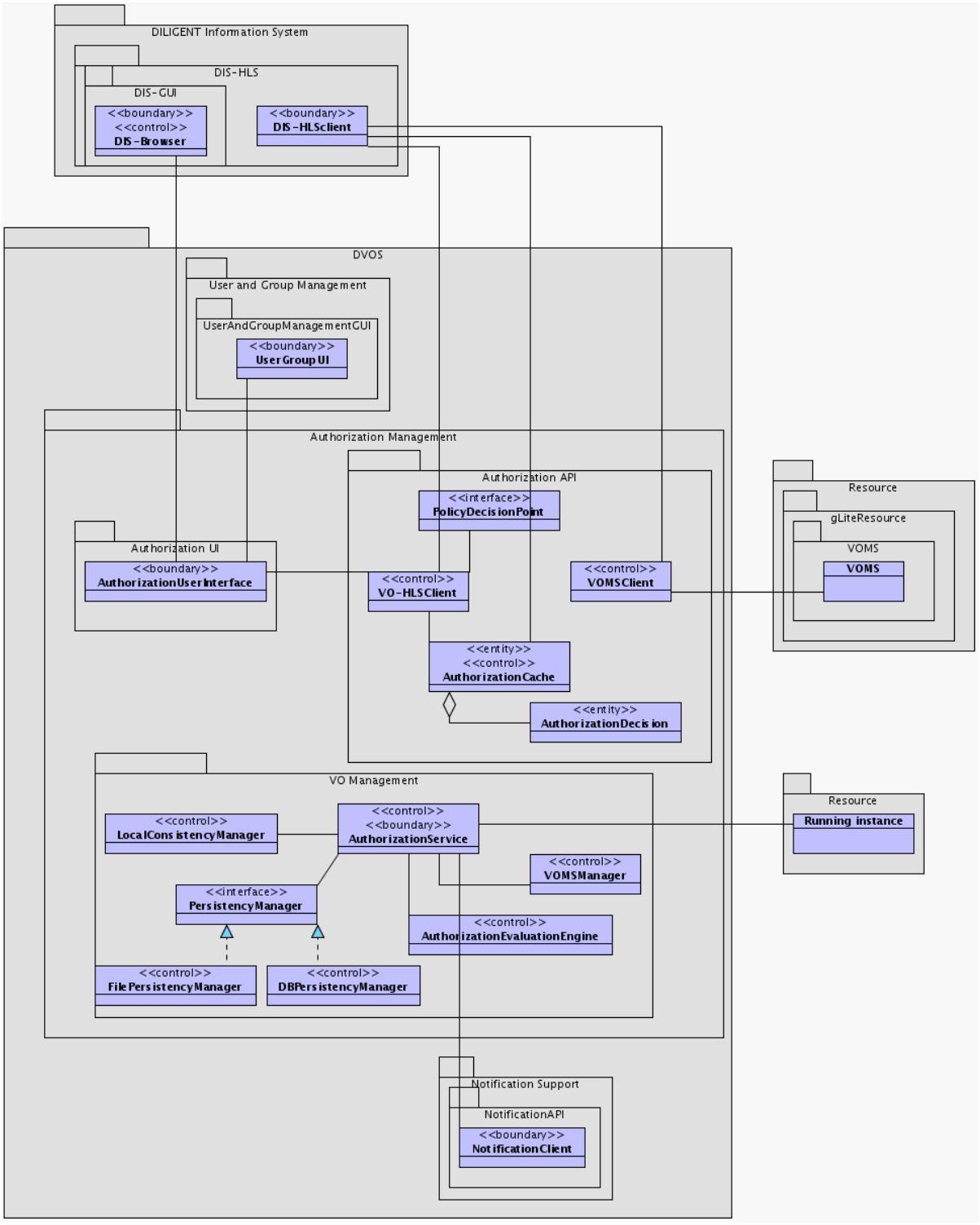

Figure 34. Authorization Management - Logical View

Authorization UI Package

- **AuthorizationUserInterface** - This class is a graphical user interface to VO Management logic. It allows users to manually inspect and modify VO and VO hierarchy status. This is achieved exploiting classes of the AuthorizationAPI package; it also interacts with the DIS visualization package and the User Management package to discover and select users and resources to be included in/removed from the VOs.

Authorization API Package

Classes of this package provide DILIGENT Resources with a set of API to manage DILIGENT VO-level Authorization. The aim of these classes is to hide details of the Authorization Management package to clients, allowing them to decouple their behaviour from the Authorization Management's internal structure. Classes of this package also implement a local cache. This can be used to speed up authorization decisions at the resource side.

- **VO-HLSClient** – This is the main class of the Authorization API package. It provides clients with a local view of the DILIGENT Virtual Organization hierarchy. This allows them to query and modify the VO and the VO hierarchy without the need to explicitly contact different *AuthorizationServices*. This class also allows clients to enable and disable the use of the local authorization cache. Association with the DIS-HLSClient allows the class to discover remote *AuthorizationServices* to contact. As shown in Figure 34, implementations of the PolicyDecisionPoint interface (provided by service designers) should use this class to check VO-level authorization.
- **AuthorizationCache** – This class is in charge to store and provide authorization decisions obtained by *AuthorizationServices*. DILIGENT resources using the cache can define the lifetime of information maintained in it.
- **AuthorizationDecision** – This class models the authorization decision obtained by the *AuthorizationService*. Each *AuthorizationDecision* has a limited lifetime assigned by the *AuthorizationService* that release it. The effective lifetime of the *AuthorizationDecision* is the minimum time between the lifetime assigned by the *AuthorizationService* and the lifetime defined by the DILIGENT Resource using the cache.
- **VOMSClient** – This class provides functionalities needed by DILIGENT Services in order to access gLite services. This class is in charge to locate and contact the VOMS service and to create the Proxy Certificate containing the Attribute Certificate obtained.
- **PolicyDecisionPoint** – This interface must be implemented to enforce authorization on incoming requests. The container invokes operations of this interface when a new service request is received. The default implementation provided in the DVOS package is suitable to enforce VO-level authorization on service operation only. In order to enforce service specific authorization policies DILIGENT services must provide their own implementations of this interface.

VOModel

This package represents the common base for interoperation between AuthorizationAPI and VO Management packages. It contains classes of the VO Model described in Figure 26.

VO Management

This package contains the core part of the Authorization Management functionalities introduced in Section 7. The following classes are in charge to maintain the whole status of the DILIGENT VO hierarchy and to implements VO management related behaviour.

- **AuthorizationService** – This class is in charge to maintain and provide authorization information related to a single VO (i.e. Users, Roles, Permissions and Sharing Rules of that VO). Each *AuthorizationService* represents a single VO in the DILIGENT VO hierarchy and, when asked, it supplies a "VO-local" view of this information. Information returned by an *AuthorizationService* is related to the position of that VO in the hierarchy. Moreover, this "VO-local" view depends on the type of information requested. E.g. asking for Users belonging to a VO will return all the users of that VO, including all the users of the sub-VOs. Asking for Roles available in the VO, instead, will return all the Roles defined in that VO, including all

the roles inherited from super-VOs[24]. Every time a new VO is created, a new *AuthorizationService*, representing the new VO, must be instantiated. This task is accomplished with the support of the *ServiceDeployer* class provided by the Keeper service (see Section 6.3). *AuthorizationService* also interacts with *NotificationCollector* during VO removal operation in order to Notify *VO Managers* of the sub-VOs.

- **AuthorizationEvaluationEngine** – This class is used to generate an *AuthorizationDecision* basing on the status of the VO. Requests for authorization decision coming from *VO-HLSClients* are resolved using instances of this class. Encapsulation of the evaluation engine in a separate class allows to easily extend the VO Model without the need to modify the *AuthorizationService* implementation.
- **LocalConsistencyManager** – This class maintains the consistency between permissions defined by *VO Managers* and sharing rules defined by *Resource Managers* for resources available in a VO. This consistency must be enforced every time new permissions are defined in the VO. This class is also in charge to rearrange permissions defined in the VO after modification of sharing rules.
- **PersistencyManager** – This class is used by the *AuthorizationService* to manage persistency of the status of a VO. Two PersistencyManager implementations are provided by DVOS: *FilePersistencyManager* and *DBPersistencyManager*. At VO creation time, the *VO Manager* chooses the *PersistencyManager* to use.
- **FilePersistencyManager** – This implementation of *PersistencyManager* stores the VO status in the local file system. To guarantee security, VO status information is encrypted with a key provided by the *VO Manager* during VO creation. Due to its simplicity and independence from other technologies, this implementation is most suitable for small VOs with few users and resources.
- **DBPersistencyManager** – This implementation of *PersistencyManager* stores the VO status in a relational database. The database to use must be available on the machine where the *AuthorizationService* is running. Moreover, the database must be configured with the schema corresponding to the VO Model. The configuration parameters needed to access the database (i.e.: database location, port number, username, password, etc.) must be provided by the *VO Manager* during VO creation. Due to the high performance of relational databases, this implementation is most suitable for large VOs with a lot of users and resources.
- **VOMSManager** – This class is in charge to contact the VOMS administration interface to manage the VOMS VOs associated to an *AuthorizationService*. This class allows maintaining consistency between the set of users of the *AuthorizationService* and that of the VOMS VOs. As described in the Authorization Management Design Considerations (see Section 7.2.4.1), the VOMS VO associated to the DILIGENT VO must be manually created. The *AuthorizationService* automatically creates an instance of this class when a VOMS VO is associated to it.

To better explain how classes and objects introduced in the logical view interact, a communication diagram is also provided. Figure 35 shows logical layers of the DILIGENT Authorization Architecture. The diagram refers to a typical Digital Library communication scenario where a DL, named DL1, is part of the ARTE community. Entities contained in each layer are described afterward.

[24] For sub-VO inheritance rules see Section 3.7 of D1.1.1 Test-bed Functional Specification.

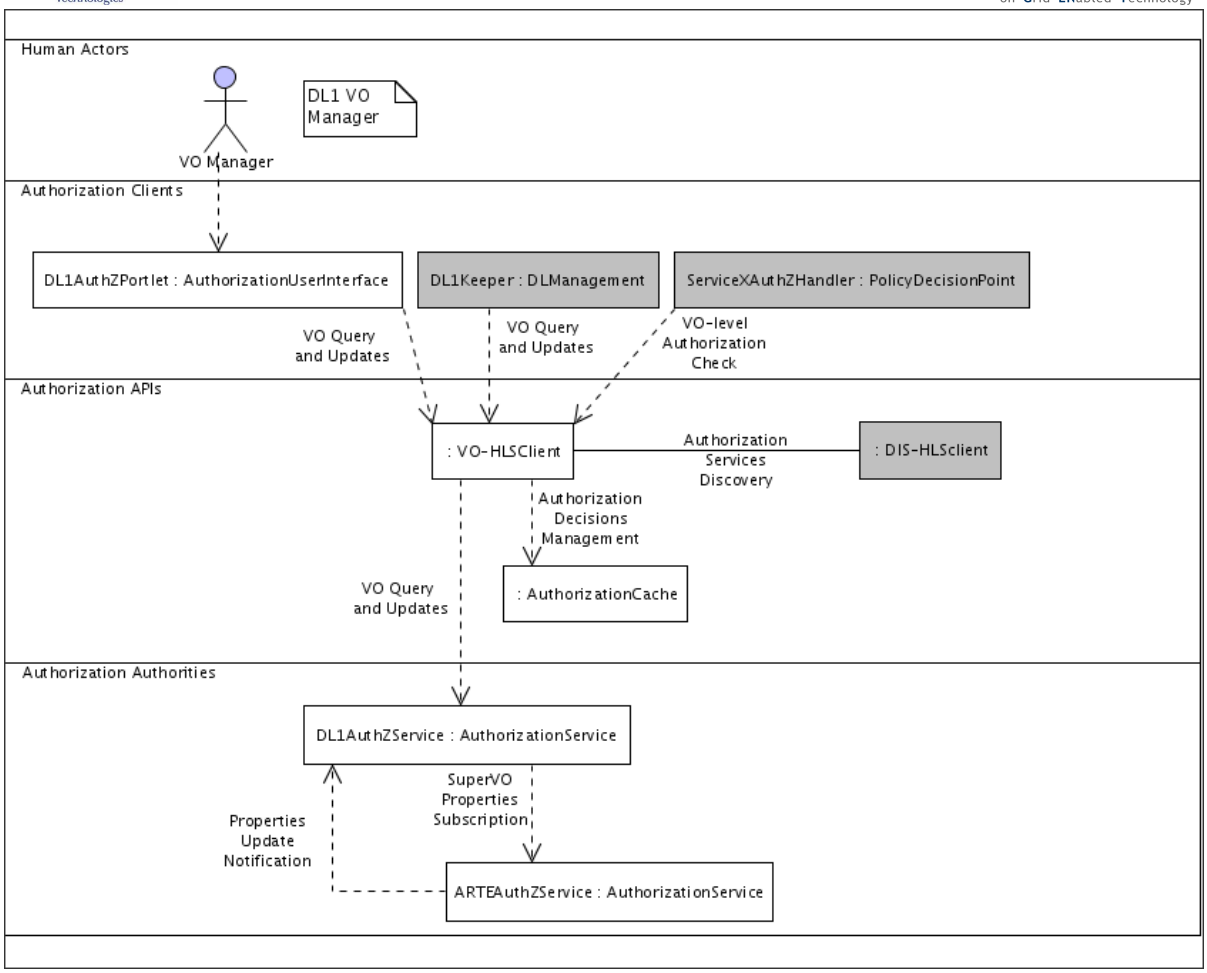

Figure 35. Authorization Management - Global Authorization Architecture

Authorization Authority Layer: the lower layer contains *AuthorizationService* instances representing the Authorization Authorities for ARTE and DL1 VOs. DL1 *AuthorizationService* acts as observer of the ARTE *AuthorizationService*. When a modification occurs in the ARTE VO then the DL1 *AuthorizationService* is notified.

Authorization APIs Layer: this layer contains the authorization library that hides details of the underlying services to clients. The *VO-HLSClient* object shown in the diagram is in charge to discover and contact *AuthorizationService instances* to perform operations requested by clients. It is also responsible to manage *AuthorizationCache* objects. Each Authorization Client using the Authorization APIs has its own private cache.

Authorization Clients Layer: in this layer three possible clients using the Authorization APIs library are shown. *DL1AuthZPortlet* is the authorization portlet that allows the DL1 *VO Manager* to manage DL1 authorizations. DL1Keeper is the *DLManagement* service in charge to deploy and monitor DL1 services (see 6.5.2.1). *ServiceXAuthZHandler* is an authorization handler of a DILIGENT Service included in the DL1 VO; the designer of service X is in charge to provide this handler.

7.2.3 Deployment View

The deployment diagram in Figure 36 depicts a possible configuration for *AuthorizationServices*, *AuthorizationUserInterfaces* and *Authorization API*. In this diagram two VOs are represented: the *Arte VO* and a sub VO named *DL1*.

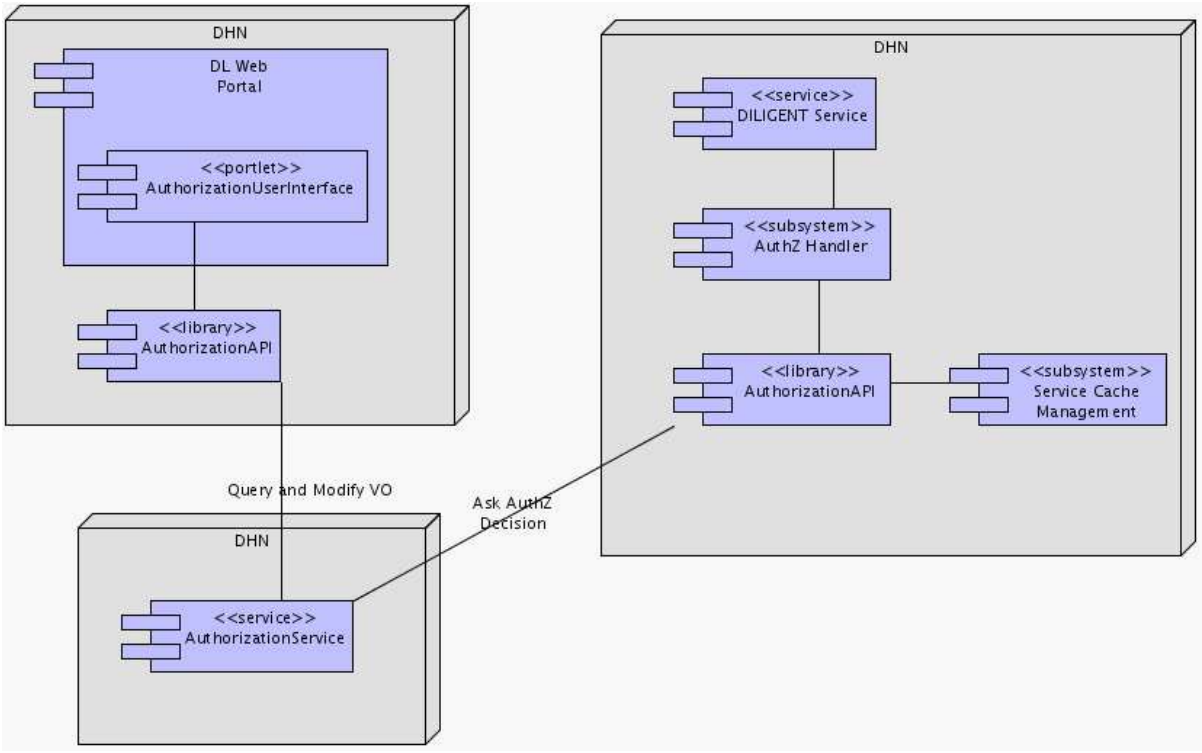

Figure 36. Authorization Management - Deployment View

Each component is briefly described afterward.

- **AuthorizationUserInterface** – In order to access and manage a *VO AuthorizationService* a corresponding *AuthorizationUserInterface* portlet must be available in a Web portal. The DL Web Portal itself hosts portlet for the VO modelling the DL. For other VOs (e.g.: ARTE VO, ImpECt VO) the DILIGENT Web Portal hosts the corresponding AuthorizationUserInterface portlets. The *AuthorizationUserInterface* uses functionalities provided by the Authorization API library.
- **AuthorizationAPI** – this library hide to clients (e.g.: DILIGENT Resources, AuthorizationUserInterfaces or authorization handlers) the internal structure of the Authorization Management package. It allows clients to easily contact *AuthorizationServices*. This library is deployed on each DHN including portal nodes.
- **AuthorizationService** – This is a WSRF Service representing a single VO. A new instance of this service is deployed when a new VO is created. These services are in charge to maintain the authorization status of the DILIGENT VO hierarchy. This status can be inspected and modified through functionalities of the *AuthorizationAPI* library.

7.2.4 Service design

7.2.4.1 Design Considerations

DILIGENT Authorization architecture considerations

In the DILIGENT infrastructure Authorization Authorities (AA), entities in charge to manage authorization information, are distributed over DILIGENT nodes. This solution avoids performance issues due to a centralized approach, but introduces some disadvantages: mainly the need to manage interactions among Authorization Authorities, and the need for DILIGENT resources to know which AA must be asked for authorization decisions.

The former problem is solved federating Authorization Authorities in a hierarchical structure, and adopting a push model to propagate VO status changes in the sub-tree rooted at that VO. This model has been adopted to reduce the time to resolve query on *AuthorizationServices*. The propagation delay introduced by this approach must be carefully considered when a client interacts with multiple VO in the same branch of the VO hierarchy.

The latter problem could be solved by propagating security information to nodes hosting DILIGENT resources. This information could be pushed to a local (to the node) Authorization Authority and used to take local authorization decisions. Therefore, resources hosted by a node have a local access point to ask for these decisions. Nevertheless, the dissemination of authorization information with the model explained still has many open issues. The main problems deal with: (i) the security level essentially offered by each node, (ii) the recover after distributed faults of local Authorization Authorities, and (iii) the duplication of authorization information over network nodes. Moreover, models for authorization management allowing specifying rule based authorization policies[25] don't fit very well with the dissemination of the authorization information, because often the selection of nodes where to propagate authorization information is a complex task.

For these reasons the retrieval of authorization decision adopts the pull model. In order to notify resources about where to ask for an authorization decision, the name of the Authorization Authority to contact is included in the Proxy Certificate of the user submitting a request. This information can be extracted from the certificate using the Authentication APIs provided by DVOS (see 7.1.4.2.3).

To speed up the retrieval of authorization decisions each DILIGENT Resource may use a local cache. The use of the cache improves performances during authorization checks.

VO model considerations

With regard to the Authorization model the XACML standard has been deeply inspected. It offers a very flexible and powerful language to specify authorization policies, but its adoption has been discarded as a global authorization model for performances issues. The evaluation of XACML policies to come to an authorization decision is a computationally intensive task to be performed for every service invocation. The adoption of this model, instead, is recommended for the management of local policies due to the low number of policies typically needed by resource (or service) specific authorization models. The DILIGENT VO Model adopted is explained in Section 7.

The design of the authorization management package is as much as possible independent from the DILIGENT VO model. This allows to easily expand and modify the VO model without the need to change services defined in the authorization architecture.

Consideration about existing Authorization Authorities implementation

Some services have been implemented to act as Authorization Authorities in a distributed environment. Mainly the *Virtual Organization Membership Service* (VOMS) [30] and the *PrivilEge and Role Management Infrastructure Standards Validation* (PERMIS) [31] have been evaluated as repositories of authorization information. Their use has been discarded mainly because of the impossibility to dynamically deploy instances of these services. Moreover, the VOMS maintain information about users and groups, but not about resources. This makes it very difficult to map the DILIGENT VO model onto the VOMS VO model. The PERMIS VO model is very similar to the DILIGENT VO model, but this service does not allow a hierarchical structure for the Virtual Organization (as DILIGENT requires).

[25] XACML is an example of that language.

The solution provided by the DILIGENT *AuthorizationService* allows to dynamically deploy new Authorization Authorities for the new Sub-VO created, using the local file system to persist the VO status. This solution has the disadvantage to bind the security of the Authorization Authority to that offered by the DHN hosting the service. Another problem is that every service running on the same DHN has the same access to the local file system, thus the status of the VO persisted by the *AuthorizationService* must be protected. For this purpose, it is encrypted using a key provided by the VO Manager. This solution significantly decreases integrity problem (as long as the key is kept secure and periodically changed) but does not assure availability of the VO status. Information stored can be always deleted by hostile services running on the same nodes. The database persistency option solves availability and integrity issues, but requires an exiting database able to store the status of the VO. The database schema to use will be provided together with the corresponding persistency option.

Considerations about interaction with gLite authorization infrastructure

Another important authorization issue is related to the management of the VOMS of the DILIGENT infrastructure. When attempting to define interactions between DILIGENT Authorization Management and VOMS two main limitations must be taken into account: some gLite services, at the moment, only allow specifying access control on a VO basis, not on a group basis, and updates in authorization configuration files on some gLite services require a restart of that services.

These limitations lead to the practical impossibility to dynamically reconfigure gLite authorization when new Sub-VOs are created in the DILIGENT infrastructure. In the default scenario the authorization granularity at the gLite level is limited to the whole DILIGENT VO. The interaction scenario consists of pushing users (and services identities) of the DILIGENT VO in a corresponding VO managed by a VOMS server. With this solution, all DILIGENT services and users are entitled to use the middleware with the same authorization settings.

If finer authorization granularity is needed it must be manually managed. In those cases, different VOMS VOs must be created and linked to the corresponding DILIGENT Sub-VOs. Moreover, gLite services must be manually reconfigured in order to trust these new VOs.

7.2.4.2 Components

7.2.4.2.1 AuthorizationService

7.2.4.2.1.1 State description

The *AuthorizationService* internal structure is shown in Figure 37. The singleton pattern has been adopted to manage the WS-Resource associated to each service instance. Only one instance of *VOResource* is created and associated to each *AuthorizationService*. The *VOResource* is used to store and retrieve the status of the VO managed by the *AuthorizationService*. If the VO has a parent VO[26] then a *VOProxy* object is used to store a local copy of information managed by the parent VO *AuthorizationService*. Only information about users and resources included in the sub-VO is stored in this copy. Updates to *VOResource* or to the *VOProxy* are forwarded to the sub-VOs using the WS-Notification framework. The *AuthorizationService* portType extends the *NotificationProducer* portType providing a Subscribe operation to allow registration of *VOProxy* objects of sub-VOs. *VOResource* and *VOProxy*, if needed, are created at service start-up by the *AuthorizationHome* object.

[26] Only the DILIGENT VO, root of the VO hierarchy, does not have a parent.

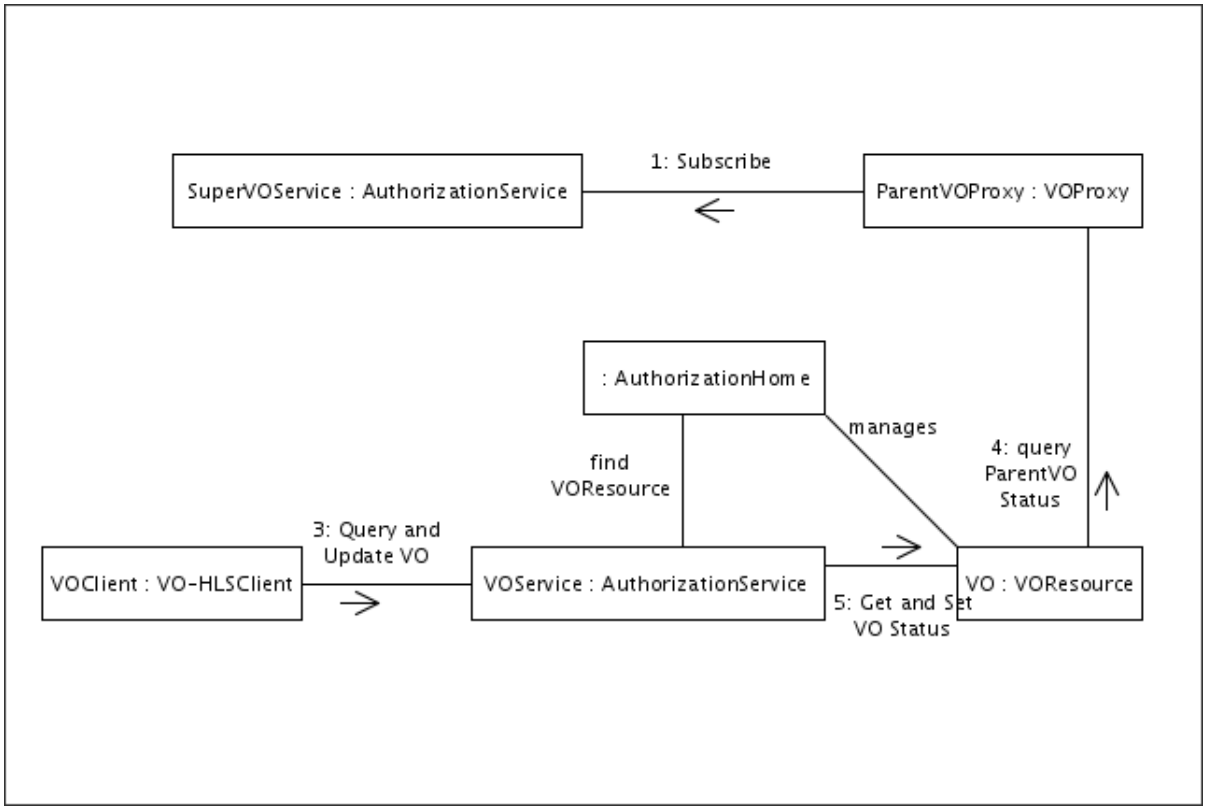

Figure 37. Authorization Management - AuthorizationService structure

7.2.4.2.1.2 Operations

Due to the high number of operation provided by the AuthorizationService, only significant operations are described here.

- **createSubVO**(subVOInformation)::ResourceId – This operation is used to create a new sub-VO of that VO. The profile of the new VO (including its name and the initial set of available DHN nodes) must be provided. The operation returns the ResourceId of the new *AuthorizationService* created for the sub-VO. A detailed description about the creation of a new VO can be found in Section 6.5.2.1.
- **removeSubVO**(ResourceId) – This operation remove the sub-VO identified through the ResourceId parameter from that VO. The sub-VO is not deleted in order to allow for the preservation of its content. *VO Managers* of the sub-VO are notified to actually delete the sub-VO.
- **addResource**(ResourceId) – This operation is used to include a resource in the VO. The resource must be already included in the super-VO.
- **removeResource**(ResourceId) – This operation is used to remove a resource from the VO.
- **addResourceAction**(ResourceId, ActionId) – This operation is used to add a new action for a resource in the VO. Actions can be real operations provided by the resource or logical action that users can perform on the resource (e.g.: *read* and *write* operation for a file). The action for that resource must be already included in the super-VO.
- **removeResourceAction**(ResourceId, ActionId) - This operation is used to remove an action from those available for the resource. All permissions related to that action are also removed.

- **addUser**(UserId) - This operation is used to include a user in the VO. The user must be already included in the super-VO.
- **removeUser**(UserId) – This operation is used to remove a user from the VO.
- **createRole**(RoleInformation)::RoleId - This operation is used to create a new role in the VO. The profile of the new role (including its name and its position in the role hierarchy) must be provided. The operation return the RoleId of the new Role created.
- **deleteRole**(RoleId) - This operation is used to delete a role from those defined in the VO. All the sub-roles rooted at that role in the role hierarchy are also removed.
- **assignRoleToUser**(RoleId, UserId) – This operation is used to assign a role to a user. Both the role and the user must exist in the VO.
- **unassignRoleToUser**(RoleId, UserId) – This operation is used to deprive a user of the specified role. Both the role and the user must exist in the VO and, moreover, the user must have been previously assigned to that role.
- **createPermission**(PermissionInformation)::PermissionId – This operation is used to create a new permission. Information of the new permission to create (including the resource and the operation involved) must be provided. The operation returns the PermissionId of the new Permission created.
- **deletePermission**(PermissionId) - This operation is used to delete a permission from those defined in the VO. The permission is also deleted from all the roles associated with it.
- **createPermissionTemplate**(PermissionTemplateInformation,ResourceId)::PermissionTemplateId – This operation is used to create a new permission template for a package. The package must be already included in the VO. Information about the new permission template to create (including the role and the operation involved) must be provided. The operation returns the PermissionTemplateId of the new PermissionTemplate created.
- **deletePermissionTemplate**(PermissionTemplateId) - This operation is used to delete a permission template from those defined for the package in the VO.
- **assignPermissionToRole**(PermissionId, RoleId) – This operation is used to assign a permission to a role. Both the role and the permission must exist in the VO.
- **unassignPermissionToRole**(PermissionId, RoleId) – This operation is used to deprive a role of the specified permission. Both the role and the permission must exist in the VO.
- **createSharingRule**(SharingRuleInformation)::SharingRuleId – This operation is used to create a new sharing rule. Information of the new sharing rule to create (including the resource and the operation involved) must be provided. The operation returns the SharingRuleId of the new sharing rule created.
- **deleteSharingRule**(SharingRuleId) - This operation is used to delete a sharing rule from those defined in the VO. All the permissions in the VO are rearranged to guarantee consistency with the reduced sharing rule set.
- **createSharingRuleTemplate**(SharingRuleTemplateInformation,ResourceId)::SharingRuleTemplateId – This operation is used to create a new sharing rule template for a package already included in the VO. Information about the new sharing rule template to create (including allowed operations) must be provided. The operation returns the SharingRuleTemplateId of the new sharing rule template created.
- **deleteSharingRuleTemplate**(SsharingRuleTemplateId) - This operation is used to delete a sharing rule template from those defined for a package in the VO. All the

permissions templates for that package in the VO are rearranged to grant consistency with the reduced sharing rule template set.

- **getAuthorizationDecision**(expression)::AuthorizationDecision – This operation is used to obtain an authorization decision. The AuthorizationEvaluationEngine evaluates the expression provided in order to return the decision.
- **setAuthorizationDecisionLifetime**(Lifetime) – This operation is used to set the lifetime of authorization decisions provided by this service.
- **subscribe()** – This operation is used by *VOProxy* objects of sub-VOs to subscribe to properties of this VO.
- **setAssociatedVOMSVO**(VOName, VOMSLocation) – This operation is used to associate a VOMS VO to this Authorization Service. When a new user is included in the VO using the *addUser(…)* operation he will be automatically added to the VOMS VO also. Similarly, when a user is removed using the *removeUser(…)* operation he will be automatically removed to the VOMS VO also. This is achieved using an instance of the VOMSManager class.

7.2.4.2.1.3 Profile description

The profile of each VO registered in the DIS includes the following information:

- **VOName**: The name of the VO.

The security profile of the *AuthorizationService* is configured as follows:

- **AuthorizationServiceSecurityDescriptor1**
  - **Authorization Chain**: A particular authorization handler is used. This handler is installed dynamically by the service during service initialization and enforces authorization based on the VO status itself.
    - **Authorization Criteria**: Only *Resource Managers* can perform the *createSharingRule(…)*, the *deleteSharingRule(…)*, the *createSharingRuleTemplate(…)* and the *deleteSharingRuleTemplate(…)* operations. Only *VO Managers* can perform all other operations of the service.
  - **Default Settings**
    - **Identity Scenario**: Service
    - **Authentication Methods**:
      - **GSISecureMessage** (integrity)

7.2.4.2.1.4 Status Description

This information is provided by the service as Resource Properties in order to enable its usage as notification topics. Nevertheless, this information is not published in the DIS for privacy reasons.

- **SubVOList:** The list of sub-VOs of this VO.
- **ResourceList:** The list of resources included in the VO.
  - **ActionList:** For each resource the list of actions defined in the VO for that resource.
  - **PermissionTemplateList:** If the resource is a package, the list of the permission templates defined for it.
    - **PermissionTemplateInformation:** For each permission template the set of information related to it.

    - **SharingRulesTemplateList:** If the resource is a package, the list of the sharing rules templates defined for it.
        - **SharingRulesTemplateInformation:** For each sharing rule template the set of information related to it.
- **UsersList:** The list of users included in the VO.
- **RolesTree:** The list of roles defined in the VO. Properties of this type can be nested to represent the whole structure of roles.
    - **RoleInformation:** For each role the set of information related to it
- **PermissionList:** The list of permissions defined in the VO
    - **PermissionInformation:** For each permission the set of information related to it.
- **SharingRuleList:** The list of sharing rules defined in the VO
    - **SharingRuleInformation:** for each sharing rule the set of information related to it
- **AuthorizationDecisionLifetime**: the lifetime of *AuthorizationDecisions* released by the service.
- **AssociatedVOMSVO**: the name of the VOMS VO associated to this VO.
    - **VOMSLocation**: the location of the VOMS hosting the associated VO.

7.2.4.2.1.5 Dependencies & Requirements

- This service relies on the following entities:

DILIGENT services

- DVOS: NotificationCollector
- Keeper: DLManagement
- DIS: DIS-HLSClient

gLite services

VOMS

Other technologies

NONE

7.2.4.2.2 Authorization API

Library component.

[to be provided in the D.1.2.3]

7.2.4.2.2.1 Description

[to be provided in the D.1.2.3]

7.2.4.2.2.2 Usage

[to be provided in the D.1.2.3]

7.2.4.2.3 AuthorizationUserInterface

Portlet component.

[to be provided in the D.1.2.3]

7.2.4.2.3.1 Operations

[to be provided in the D.1.2.3]

7.2.4.2.3.2 Interaction with other components

[to be provided in the D.1.2.3]

7.2.4.3 Deployment scenario(s)

[to be provided in the D.1.2.3]

7.3 Notification Service

The aim of the DILIGENT Notification Service is not to provide a general mechanism to manage events occurring in the system. Such a general mechanism is already provided by the WS Notification [29] implementation of the Java WS Core. The twofold purpose in designing the DILIGENT Notification Service is, instead, to allow DILIGENT users to be informed about relevant[27] changes in the system and suggest them options available as a consequence of these changes.

The Functional Specification D1.1.1 identifies three functionalities related to notification area:

- **Notify a User** – This functionality enables a DILIGENT Service to notify a DILIGENT user based on its own identity.
- **Notify a Group** – This functionality enables a generic DILIGENT Service to notify all DILIGENT users belonging to a given group.
- **Notify a Role** – This functionality enables the notification of all users having a particular Role.

7.3.1 Use-Case View

The D1.1.1 identifies a DILIGENT Service as the publisher of notification messages. A human actor always consumes these notifications. For this reason the DILIGENT Notification Service is based on the Producer/Consumer pattern. DILIGENT Services act as notification producers. Consumers can be DILIGENT users asking for pending notifications (pull model) or a dispatcher in charge to forward pending requests to user's mailboxes (push model).

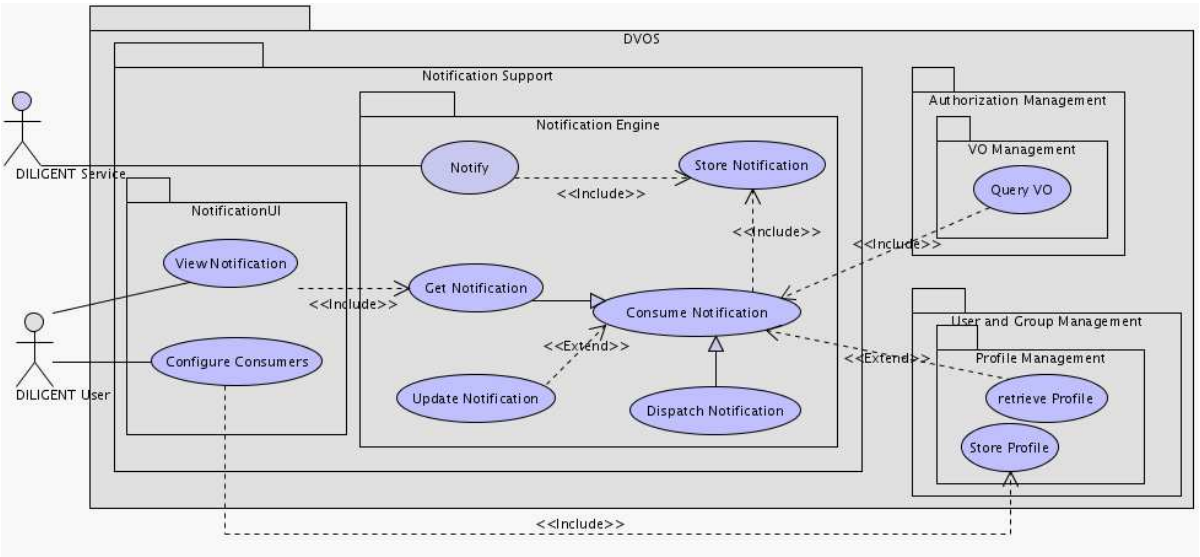

Figure 38. Notification Support - Use-Case View

[27] The term "relevant" is used to identify a subsection of the changes in the system that DILIGENT designers think to be important for DILIGENT users.

The above use-case diagram shows two main packages for this service: a *NotificationUI* package, modelling the interaction with users, and a *NotificationEngine* package, which is responsible of managing the lifetime of a notification message.

Notification Engine Package

This package is the core part of the DILIGENT Notification Service. It contains functionalities related to notification creation and dispatching.

- **Notify** - This functionality enables a DILIGENT Service to create a new notification. As mentioned above, the target of a notification can be a single user, a group of users or the set of users with a given role. Each notification will be modelled as a single message direct to the user, group or role selected.
- **Store Notification** - After its creation a notification message needs to be stored waiting for its consumption. This step is peculiar when notification delivery is performed adopting the pull model.
- **Consume Notification** - This functionality models the generic delivery of a notification message to its recipient(s), either adopting a push or pull model. Just in case of group notification or role notifications, this functionality needs to know group members and users with the given role respectively. For such a purpose, *QueryVO* functionality from *Authorization Management* and *GetGroupProfile* functionality from *Users Management* are also exploited.
- **Dispatch Notification** – This functionality models a delivery action based on a push model (e.g. email).
- **Get Notification** - It models a delivery action based on a pull model (e.g. by using a web-based client application). The NotificationUI package provides an interface allowing DILIGENT users to visualize notifications (see below).
- **Update Notification** - Once a notification has been delivered, its status can either be updated (e.g. by recording the delivery date and time) or any reference to the notification can be entirely removed from the storage.

NotificationUI Package

The NotificationUI package contains functionalities related to the User interface part of the DILIGENT Notification Service.

- **Configure Consumers** - It enables DILIGENT users to change their own delivery model. The *pull* or the *push* model can be selected. If the *push* model is selected, notifications sent to the user will be forwarded to his email address. Otherwise, if the *pull* model is selected, notifications for the user will be stored in the Notification Service waiting for user to view them. The dispatching model to use is stored in the profile of the user through the *Store Profile* functionality of the *UserAndGroupManagement* package (see Section 7.4).
- **View Notification** – When the *pull* model is selected this functionality enables DILIGENT users to view notifications stored in the DILIGENT Notification Service. Of course, each DILIGENT user will be entitled to view notification sent to him only.

7.3.2 Logical View

Functionalities described in the above Use Case section are provided by a single service: the *NotificationService*. It is in charge to receive new notifications and to manage them accordingly to the configuration set by notification receiver. This service is decomposed in classes reported in the following Logical View.

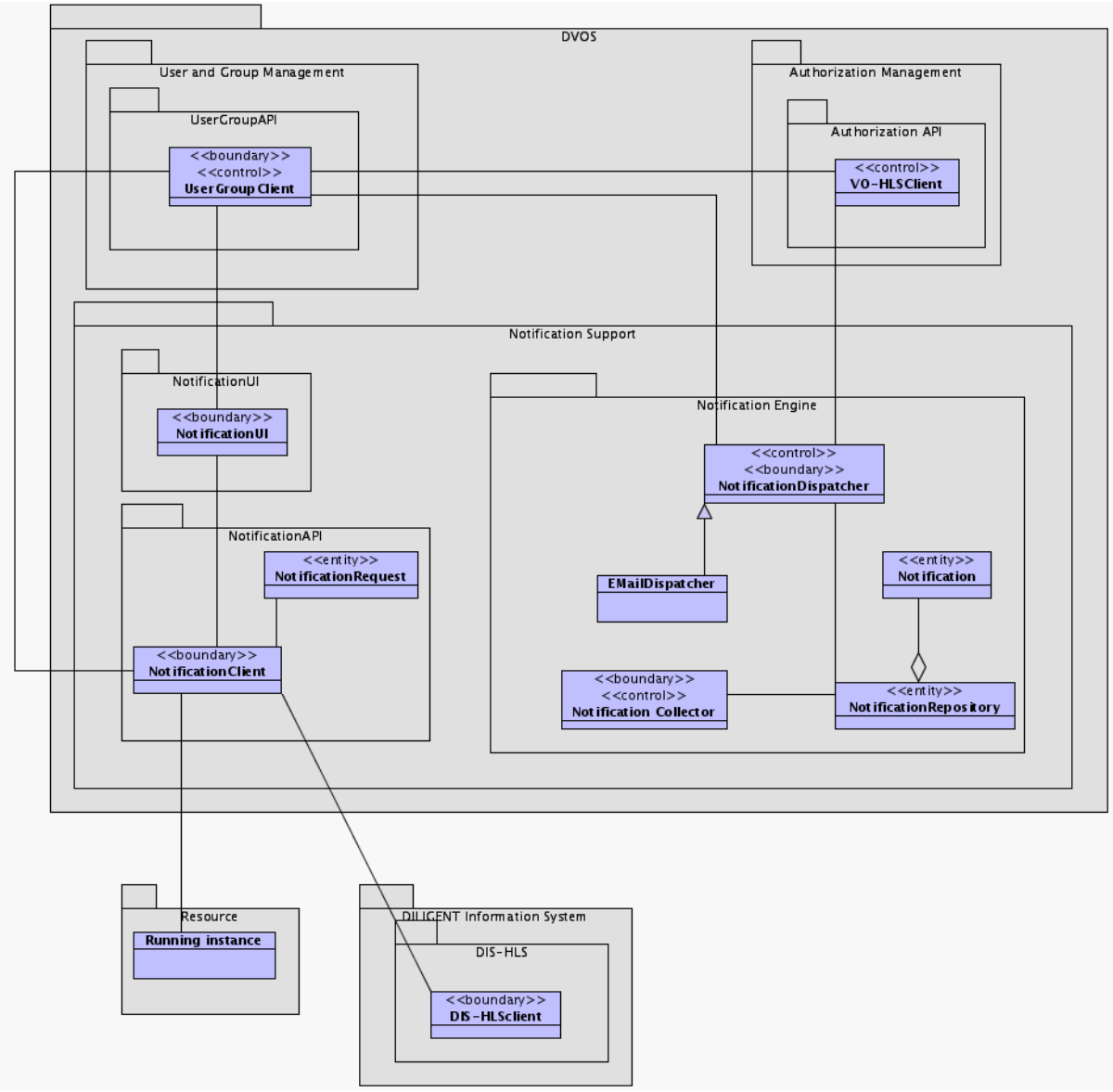

Figure 39. Notification Support - Logical View

Notification Engine package

The Notification Engine package includes classes for managing the whole notification process (NotificationCollector and NotificationDispatcher), for keeping track of not-yet-delivered messages (NotificationRepository), and for managing user-defined delivery configurations (ConfigurationRepository).

NotificationUI

The Notification UI interacts with users for both pulling notifications from the repository as well as for configuring their delivery settings (push/pull mode, email delivery, frequency of delivery, etc.)

7.3.3 Deployment View

Notification Support relies on two distinct components that can be deployed independently on different DILIGENT Hosting Nodes:

- NotificationUI is a portlet component which is hosted by the DILIGENT Portal Engine.
- Notification Service: a WSRF service hosted on a DILIGENT Hosting Node, which is in charge to implement the Notification Engine package behaviour.

To improve notification reliability, multiple Notification Services can be deployed on different hosting nodes. All of them are registered to the DILIGENT Information Service that will provide them with an endpoint reference to the NotificationUI for retrieval and configuration.

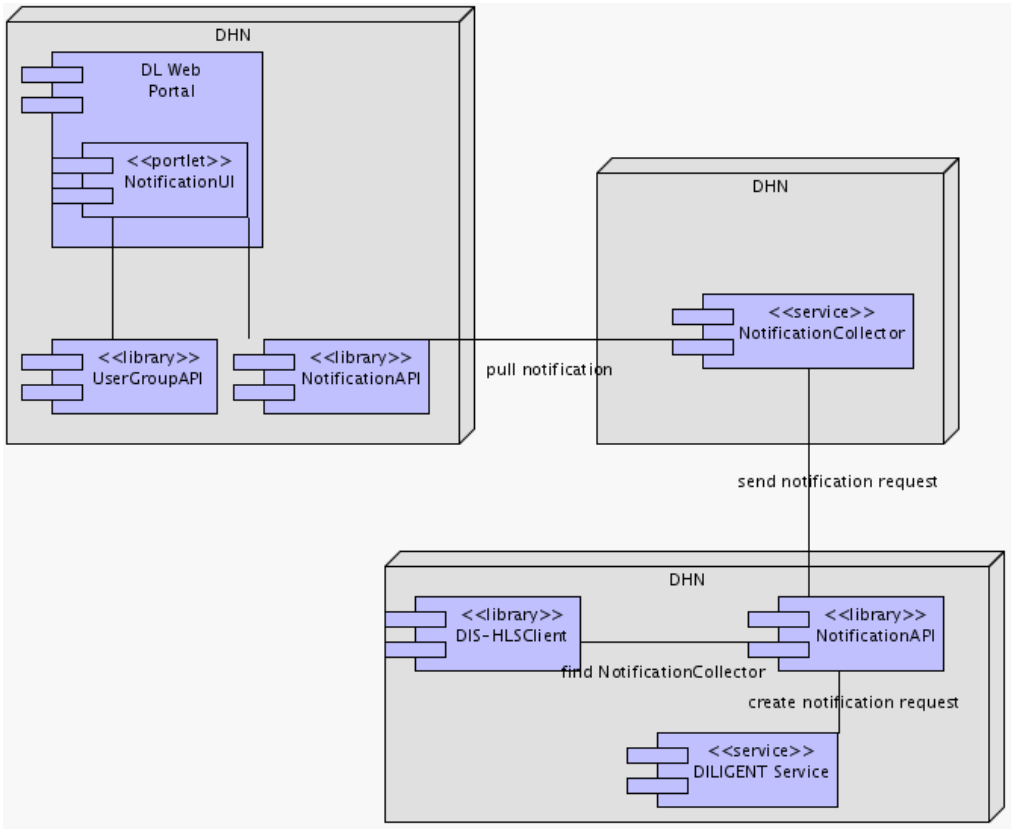

Figure 40. Notification Support - Deployment View

[to be detailed in D.1.2.3]

7.3.4 Service Design

[to be provided in D1.2.3]

7.4 User and Group Management Service

According to the D1.1.1 "Test-bed Functional Specification" [1], the management of information about users and group of users is mainly divided into four functional parts: *users management*, *groups management*, *search*, and *invitation.*

The *Users management* part covers functionalities for adding and removing users to/from DILIGENT as well as to edit user profiles and manage user rights.

The *Groups management* part includes use cases to create and remove group of users as well as to edit the group profile. The model underlying the adjunction of a user to a group implies that: (i) a user is invited to join a group, (ii) the invited user accepts or rejects the

invite, and (iii) the User Manager includes into the group the user that has accepted the invitation.

DILIGENT groups are useful to easily manage information related to a set of users, to discover communities with related interests (in order to invite them to join DLs) and to simplify management tasks through common actions on a set of users.

The *Search* part deals with searching and browsing of users and groups.

The *Invitation* part deals with invitee users and groups to join a DL, to get access to a Collection, etc. The mechanism is similar to that explained in the *Group Management* section.

7.4.1 Use-Case View

Hereafter, functionalities identified within deliverable D1.1.1 Test-bed Functional Specification are reported and grouped according to the above identified functional areas:

- Users Management
  - Add a User to DILIGENT (4.5.7)
  - Remove a User from DILIGENT (4.5.12)
- Group Management
  - Create a Group (4.5.1)
  - Remove a Group (4.5.6)
  - Add a User to a Group (4.5.4)
  - Remove a User from a Group (4.5.5)
- Invitation Support
  - Invite a User (4.5.19)
  - Invite a User to a DL (4.5.21)
  - Invite a User to a Group (4.5.22)
  - Invite a User to a Complex Object (4.5.23)
  - Invite a Group (4.5.24)
  - Invite a Group to a DL (4.5.26)
  - Invite a Group to a Complex Object (4.5.27)
  - Invite a User/Group to a DL/Complex Object
  - Propose User Rights (4.5.20)
  - Propose Group Rights (4.5.25)
  - Accept Invite
  - Request User Rights (4.5.9)
- Profile Management
  - Edit Group Profile (4.5.2)
  - Edit User Profile (4.5.8)
  - Store Group Profile (4.5.3)
  - Store User Profile (4.5.10)
  - Remove User Profile (4.5.11)
- User/Group Search
  - Select Groups (4.5.13)
  - Select Users (4.5.16)

- Browse Groups (4.5.15)
- Browse Users (4.5.18)
- Search for Groups by Details (4.5.14)
- Search for Users by Details (4.5.17)

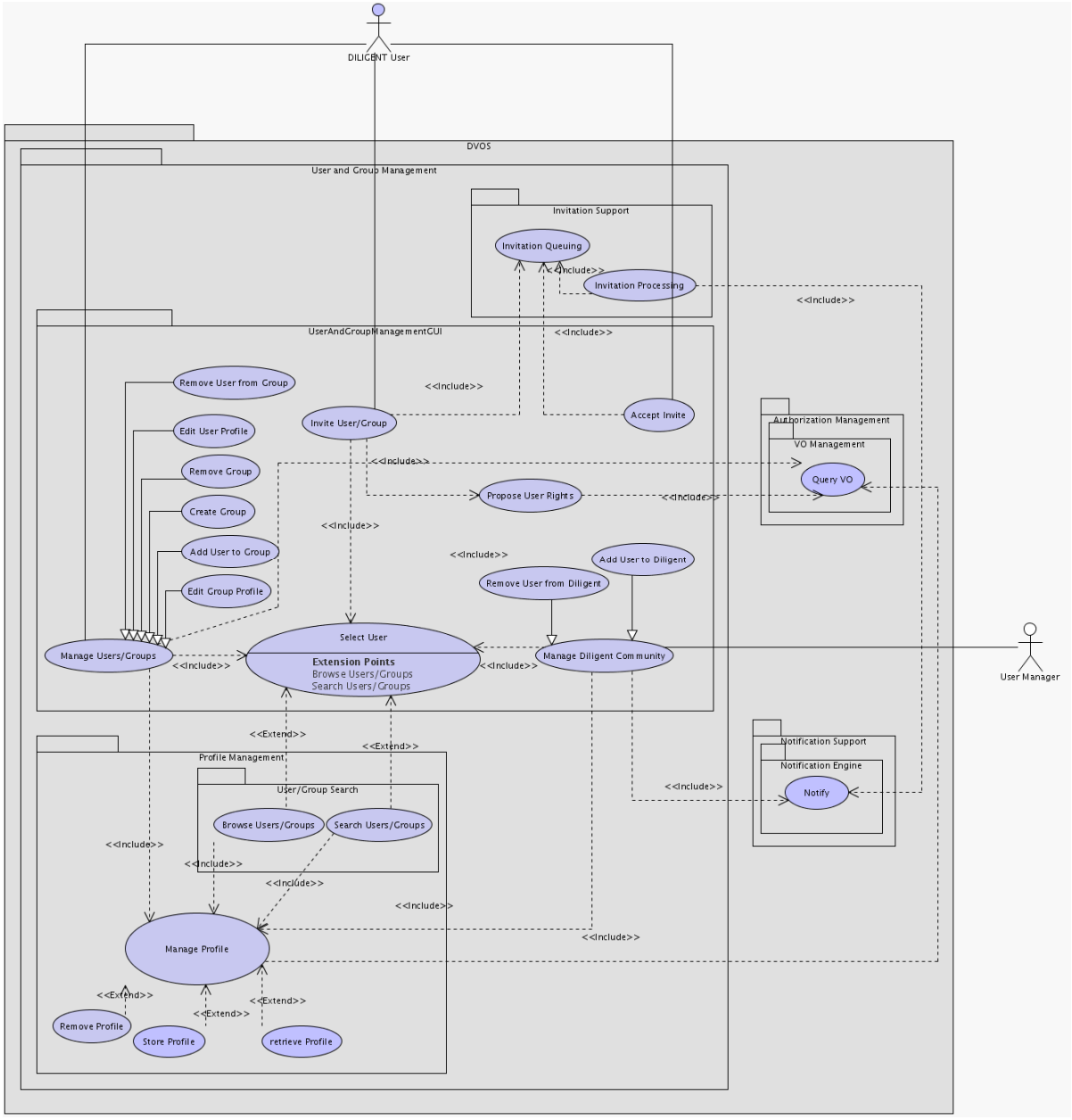

Figure 41. Users Management - Use-Case View

The above use-case diagram shows the two main actors involved in User and Group Management functionalities:

- The User Manager is mainly concerned with registration/removal of users and groups in/from the DILIGENT Community. She/He is in charge of accepting/rejecting registration requests coming from potential members and modifying user personal information (e.g. name, surname, DN, etc.)

- The DILIGENT User, on the other hand, is allowed to create her/his own groups[28], modify her/his preferences (e-mail, preferred language, interests, etc.), and invite other users to join Digital Libraries or to use resources.

User and Group Management are mainly concerned with the management and retrieval of user and group profiles respectively. For a detailed description of functionalities in this area see D1.1.1 "Test-bed Functional Specification" [1]. With respect to this document, only the Invitation Support package is further detailed.

Invite User/Group to a DL/Complex Object

This functionality enables DILIGENT Users to invite Users/Groups to join existing Digital Libraries or to acquire rights over a Complex Object (e.g. a Collection). By using the service GUI, users first identify the addressee of the invitation (either a group or another user), then, by interacting with the Authorization Support, identify what VO the addressee is invited to join or the complex object she/he's invited to use. Finally, she/he proposes the rights the addressee would have after accepting the invitation.

Invitation Processing

The invitation action performed by a generic user, anyway, is not sufficient in order to really allow some user to access an object or to join a DL. Actually, the VO manager takes this decision. Furthermore, the workflow for the invitation process consists of several steps:

- Any User invites someone (the recipient) to join a DL or to access a complex object.
- The recipient accepts/rejects the invitation.
- The VO Manager, finally, decides whether the recipient can be granted proposed rights.

Invitation Queuing

- This functionality models the recording and checkpointing of an invitation from its request, done by a user, up to its final acceptance performed by a VO Manager.

7.4.2 Logical View

User and Group Management is mainly concerned with the management of their profiles and their retrieval. The analysis of the use cases introduced above leads to the following logical decomposition:

[28] If they are authorized to perform this operation.

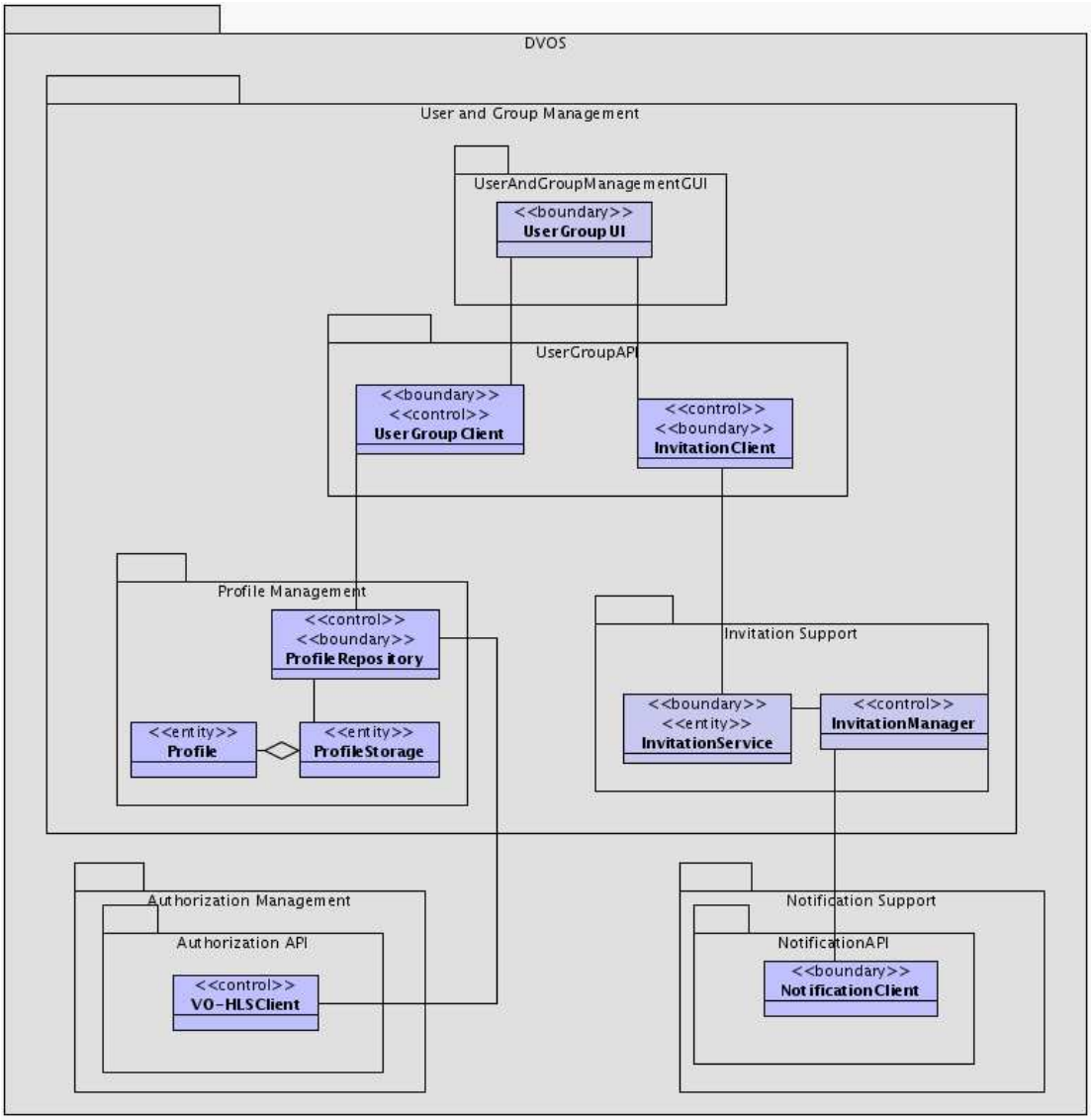

Figure 42. User and Group Management - Logical View

- The *Profile Management* package deals with the access (*ProfileRepository*) and permanent storage (*ProfileStorage*) of user and group profiles. It interacts with the *VO Management* package in order to enforce local access policies (e.g. DILIGENT users can only modify their own profile, User Managers can change user profiles only within their own VO, etc.)
- The *User and Group Management GUI* provides DILIGENT Users with an access point to the whole set of user and group-related set of functionalities.
- The *Invitation Support* package includes classes for collecting invitations made throughout the DILIGENT community (*InvitationQueue*), and managing the invitation workflow (*InvitationManager*)

7.4.3 Deployment View

The User and Group Management Service deployment scheme is organized around three components that can be deployed independently on DILIGENT Hosting Nodes:

- Invitation Service – is a WSRF service that can serve multiple VOs. Multiple instances of this service can be deployed in order to increase availability.

- Profile Repository – a component providing storage and access capabilities to user and group profiles on a per-VO basis. At least one *ProfileRepository* service needs to be deployed for each existing VO.
- UG-GUI is a portlet component and is hosted by the DILIGENT Portal Engine.

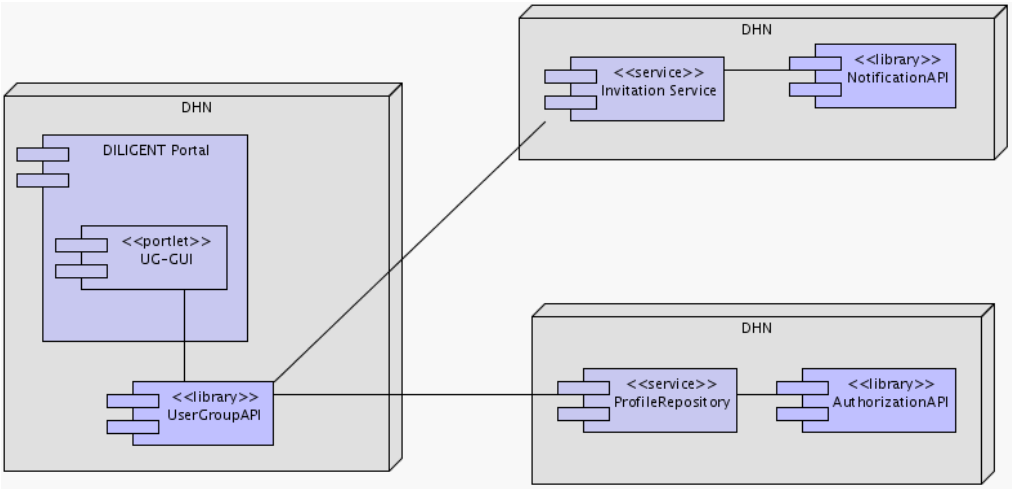

Figure 43. User and Group Management - Deployment View

[to be detailed in D.1.2.3]

7.4.4 Service Design

[to be provided in D1.2.3]

7.5 Resource Registration Support

Resource Registration Support provides mechanisms enabling resource owners to share their resources with the DILIGENT community. The DILIGENT infrastructure imposes several constraints on resource usage, where the most important is the conformity to the DILIGENT Resource Model presented in Figure 2 and the fulfillment of some sharing policies. Both registration and the definition of such sharing policies are managed through the Resource Registration Support Service by *Resource Managers*. Example of resources that can be registered are DHNs and software packages.

During a DILIGENT resource registration a Resource Manager (or a service acting as Resource Manager) must provide for each resource:

- a *profile* – an XML description of the resource conforming to the DILIGENT Resource Model[29],
- a set of *sharing rules* – rules that define the way a VO is entitled to use a resource.

7.5.1 Use-Case View

According to deliverable D1.1.1 (functional specification), for the Resource Management area, the Dynamic VO Support Service is involved in a number of functionalities:

- Add a Resource to DILIGENT (4.3.1)
- Register a Resource (4.3.2)
- Edit Sharing Rules (4.3.3)
- Remove a Resource (4.3.8)

[29] The XML Schema of the DILIGENT Resource Model will be provided in the D.1.2.3.

The following diagram includes the four above-mentioned functionalities along with a number of internal functionalities and interactions with other Collective Layer Services. Each of them is briefly described afterwards.

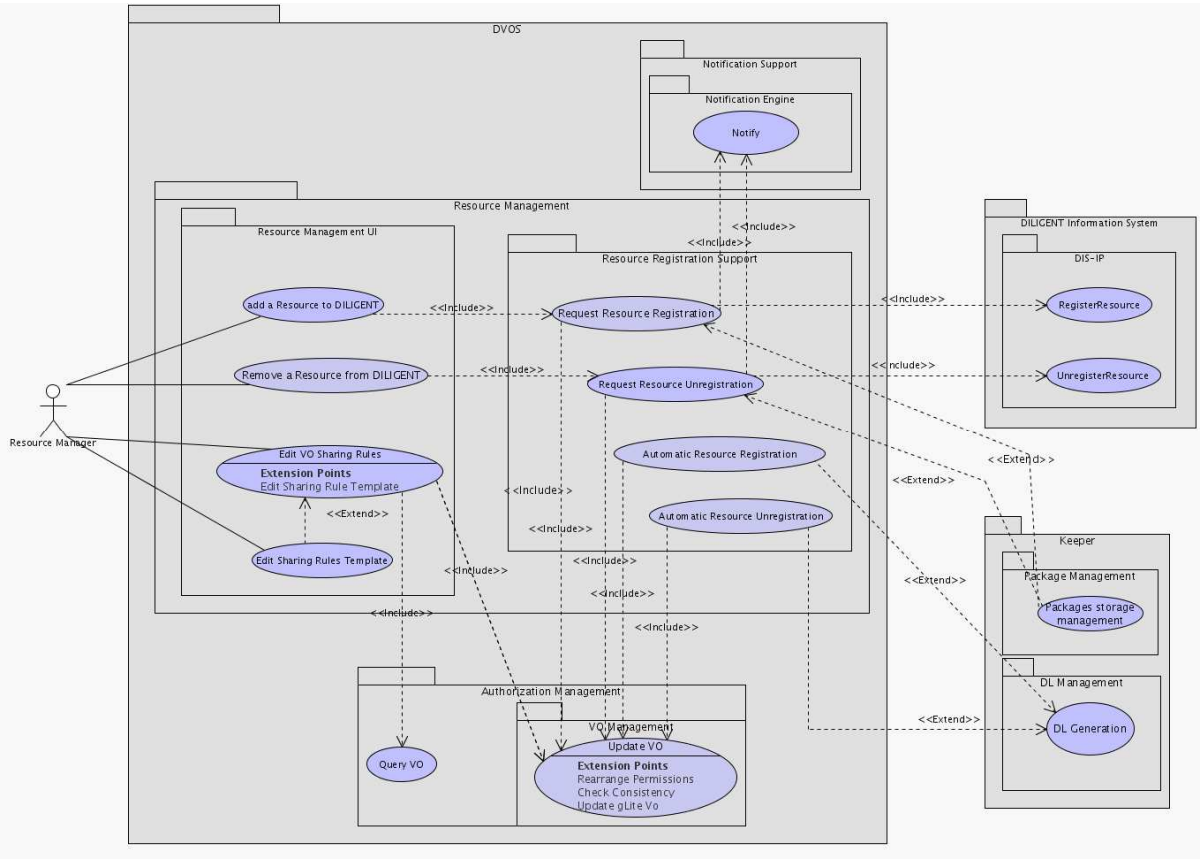

Figure 44. Resource Registration Support - Use-Case View

Resource Management UI Package

Use cases of this package cover the user interface part of Resource Management functionalities. These interfaces enable *Resource Managers* to ask for resource registration and unregistration as well as to change resource-sharing rules.

- **Add a Resource to DILIGENT** - This functionality allows *Resource Managers* to request the registration of a new resource within the DILIGENT infrastructure. Performing this use case *Resource Managers* can also describe the resource to being registered, providing its profile. The DILIGENT *VO Manager* is notified in order to add the resource to the DILIGENT VO. This functionality uses the *Request Resource Registration* use case to perform the task. Editing of sharing rules must be performed after the registration process end through the use case *Edit VO Sharing Rules*.
- **Remove a Resource from DILIGENT** - This functionality allows *Resource Managers* to ask for resource unregistration from the DILIGENT infrastructure. Interested VO Managers are notified in order to remove the resource from each VO using it. This functionality uses the *Request Resource Unregistration* use case to perform the requested task.
- **Edit Sharing Rules** - This functionality enables *Resource Managers* to define the way a whole VO is entitled to use a resource, regardless of its members. Resource Managers can change them at any time during the resource lifetime; however, this doesn't directly change permissions on the resource, rather it triggers an action on

Authorization Support Service in order to rearrange permissions accordingly (see VO Management section)

- **Edit Sharing Rules Template** – This use case allows *Resource Managers* to set sharing rules to be applied when a new RunningInstance of a package is included in a VO. When a registered package is deployed on a node, VO-level sharing rules must be set to enable it to interoperate with other DILIGENT running instances. The Keeper, as *Resource Manager* of the deployed instance, is in charge to perform these settings. The *Resource Manager* registering the package must supply the set of sharing rules to apply when an instance of that service is deployed. A very important thing to remark is that these sharing rules are different from those set on the package of the service registered in the DILIGENT infrastructure.

Resource Registration Support

This package contains core functionalities for registration and unregistration of DILIGENT resources. Two registration (and unregistration) modes are provided: *Request Mode* and *Automatic Mode*. The *Request Mode* registration (and unregistration) ends with a notification to *VO Managers* that is then in charge to perform the requested operation by updating the VOs they manage (see Section 7.2). The *Automatic Mode* registration (and unregistration) ends with the actual modification of the interested VOs. Both *Request Mode* and *Automatic Mode* registrations (and unregistration) are described in detail in Section 7.5.4.2.1.

- **Request Resource Registration** - This functionality actually models the registration process in the *Request Mode*. *Resource Managers* can perform it in order to request the registration of a new resource to the DILIGENT infrastructure.
- **Request Resource Unregistration** - This functionality actually models the unregistration process in the *Request Mode*. *Resource Managers* can perform it in order to request the unregistration of a resource from the DILIGENT infrastructure.
- **Automatic Resource Registration** – This functionality actually models the registration process in the *Automatic Mode*. Allowed services, acting as *DL Managers*, can perform it in order to add a new resource to the DILIGENT infrastructure.
- **Automatic Resource Unregistration** - This functionality actually models the unregistration process in the *Automatic Mode*. Allowed services, acting as *DL Managers*, can perform it in order to remove a resource from the DILIGENT infrastructure.

7.5.2 Logical View

Figure 45 shows main classes and components involved in resource registration and management. As to the previous use case view, the *RegistrationUI* class contains the logic related to use cases *Add a Resource to DILIGENT* and *Remove a Resource from DILIGENT*. The *SharingRulesUI* class contains logic related to *Edit VO Sharing Rules*. In the Resource Registration Support package, the *RegistrationProcess* and *UnregistrationProcess* classes and their subclasses implement the logic for the registration (and unregistration) in *Request Mode* and *Automatic Mode*. Interactions with other Collective Layer service areas are also reported. The diagram also shows the use of Resource Registration Support by a generic DILIGENT RunningInstance.

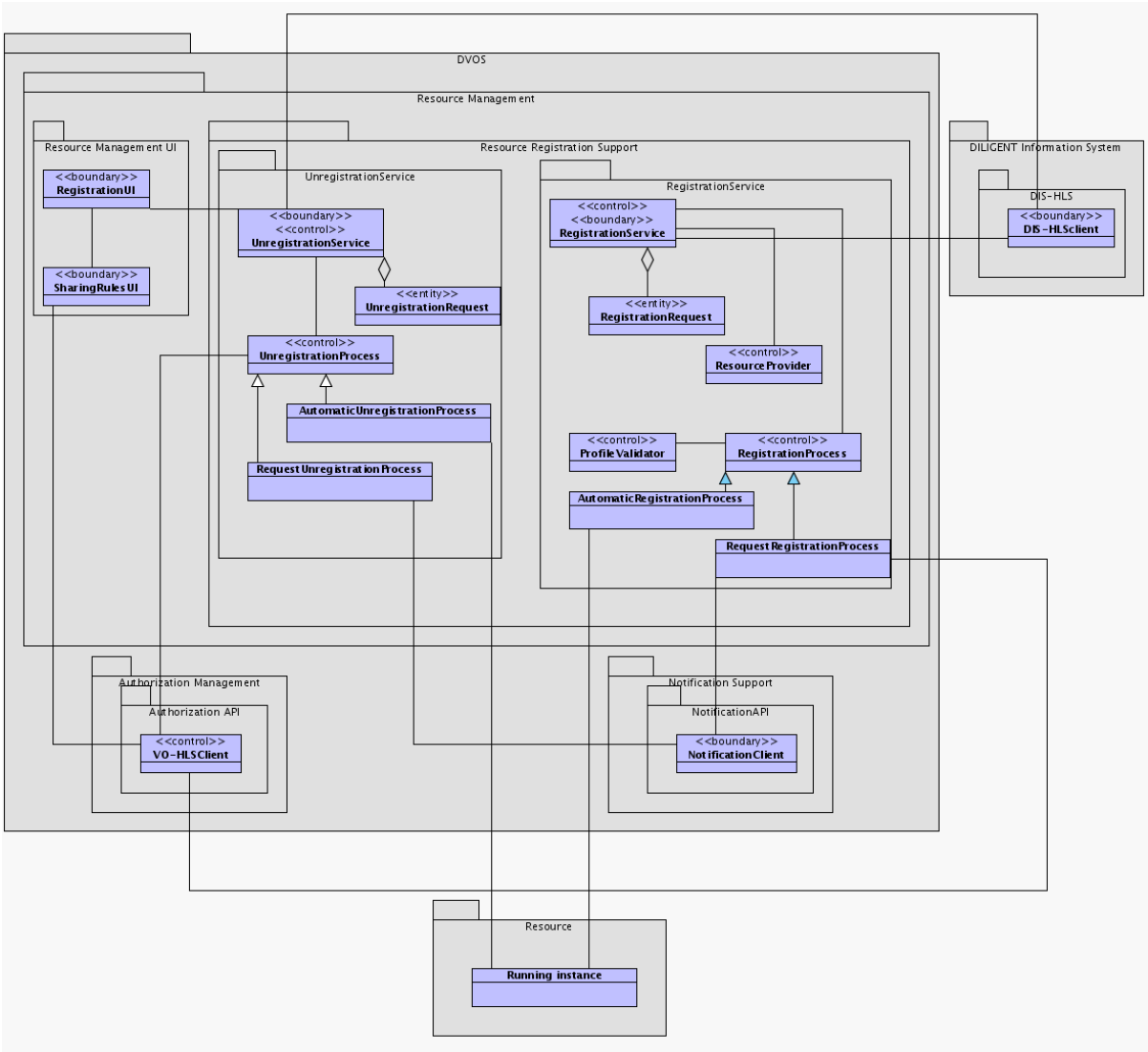

Figure 45. Resource Registration Support - Logical View

ResourceManagementUI package

- **RegistrationUI** - Through this user interface *Resource Managers* are able to register (and unregister) resources in (from) the DILIGENT infrastructure.
- **SharingRulesUI** - Through this user interface *Resource Managers* are enabled to modify sharing rules for resources previously registered in the DILIGENT infrastructure. Sharing Rules are stored in an *AuthorizationService* hierarchy and can be set on a per VO basis, allowing *Resource Managers* to grant different authorizations to different VOs in the hierarchy.

ResourceRegistrationSupport package

- **RegistrationService** and **UnregistrationService** – These are the classes providing access to the Resource Registration Support package. These classes are in charge to receive and store registration and unregistration requests sent by *RegistrationUI* class or by DILIGENT *RunningInstances*. These classes are also responsible to manage the *Process* object associated to each request.
- **RegistrationProcess, UnregistrationProcess (and their subclasses)** – These classes contain logic for *Request Mode* and *Automatic Mode* registration and unregistration processes. *RegistrationService* and *UnregistrationService* use these classes to process registration and unregistration requests respectively.

- **ProfileValidator** – This class contains logic to validate the profile of the resource being registered in the DILIGENT infrastructure. The validation of the profile is performed verifying its syntactical correctness with respect to the DILIGENT Resource Model as defined in Section 3.
- **ResourceIdProvider** – This class is in charge to generate a UniqueId for new resources registered in the DILIGENT infrastructure.
- **RegistrationRequest and UnregistrationRequest** – These classes model registration and unregistration requests managed by this package.

7.5.3 Deployment View

Figure 46 shows the deployment view of the Resource Management components.

- **RegistrationUI** is a portlet that provides *Resource Managers* with access to manual resource registration (and unregistration) provided by the *RegistrationService*.
- **SharingRulesUI** is a portlet that provides *Resource Managers* with functionalities to request inclusion of resources into VOs and to modify VO Sharing Rules.
- **RegistrationService** and **UnregistrationService** are WSRF services that provide registration and unregistration functionalities.

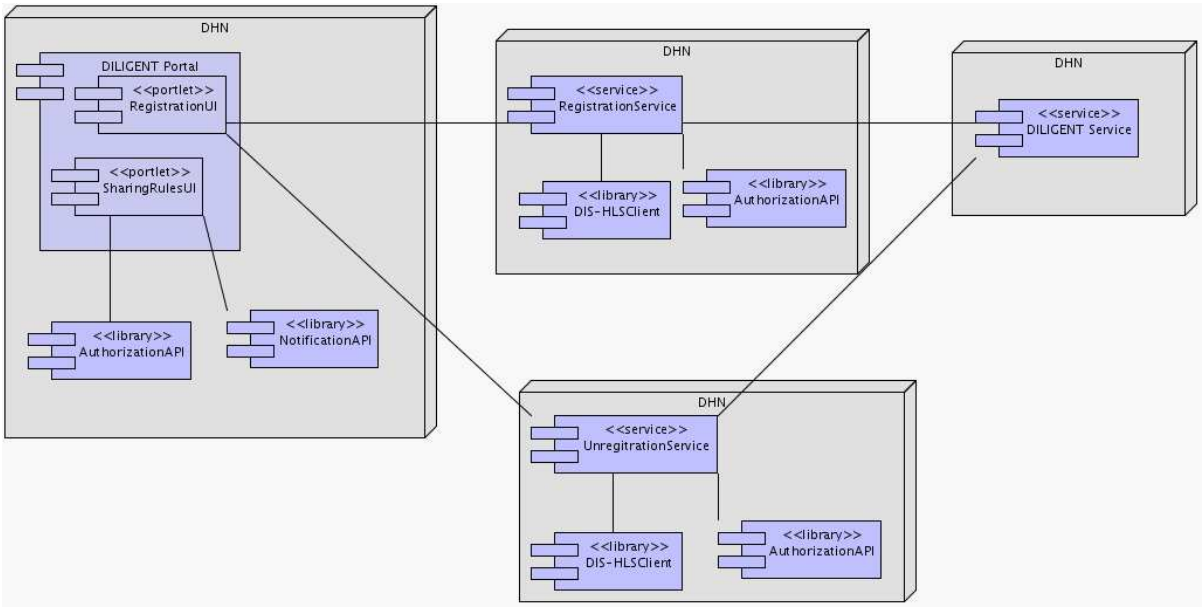

Figure 46. Resource Registration Support - Deployment View

7.5.4 Service design

7.5.4.1 Design Considerations

One issue driving the Resource Registration Support Management area design is the need to support the automatic registration of resources performed by services (acting as *DL Manager*) during the DL generation and lifetime. Registration of these resources must be performed without manual user intervention. This need has been addressed with the *Automatic Mode* registration and unregistration functionalities.

Another issue that Resource Management must address is the need to obtain the UniqueId assigned by Resource Registration Support to DILIGENT Resources before the creation of the resource itself. This avoids the need to communicate the UniqueId to the resource after

its creation[30]. A separation between obtaining a UniqueId and registering a new resource allows creators of resources to easily communicate the UniqueId as a parameter of the resource creation. UniqueIds generated by the registration service are compliant with the Universal Unique IDentifier standard [28].

7.5.4.2 Components

7.5.4.2.1 RegistrationService

7.5.4.2.1.1 State description

The pattern adopted for the *RegistrationService* design is the WS-Resource Factory Pattern. When the registration of a new resource is required, the client asks the *RegistrationFactoryService* to create a new *RegistrationResource*. *RegistrationService* uses it to maintain the status of a registration process. Each *RegistrationResource* can be used to manage the registration state of only one resource in the DILIGENT infrastructure. The RegistrationService automatically destroys each RegistrationResource at the end of the registration process, but the client can also destroy it explicitly before the end of this process. The internal structure of the RegistrationService is shown in Figure 47.

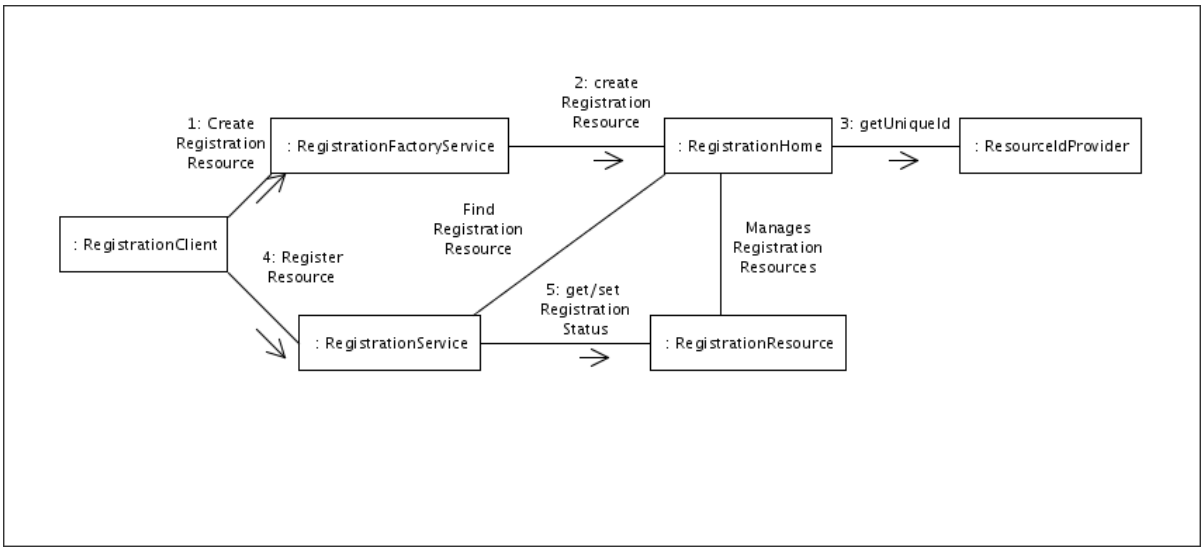

Figure 47. Resource Registration Support - RegistrationService Structure

7.5.4.2.1.2 Operations

In this paragraph operations are listed together with the service they belong to. This is achieved by placing the name of the service or that of the resource before the signature of the operation.

- **createRegistrationResource**(Mode)::EPRType – This operations allows clients to create a new WS-Resource of type RegistrationResource. Such a resource can be used to register a new resource in the DILIGENT infrastructure. The Mode parameter discriminates between the creation of a RegistrationResource operating in *Request Mode* or in *Automatic Mode*. The UniqueId associated to the resource being registered is generated during the creation of the new *RegistrationResource* and can

[30] This communication problem could be solved in some ways (e.g.: adding an operation to all the DILIGENT resources that allows to remotely set the UniqueId), but all these approaches originate other issues related to the management of the UniqueId.

be asked for by the Client before the start of the registration process. This operation returns the endpoint reference to the new registration resource.

- **getUniqueId**()::UUID – This operation allows the client to get the UniqueId that is assigned to the resource during the registration process. The UniqueId returned follows the Universal Unique IDentifier standard [28]. Figure 48 provides an example of *RegistrationResource* creation and use.

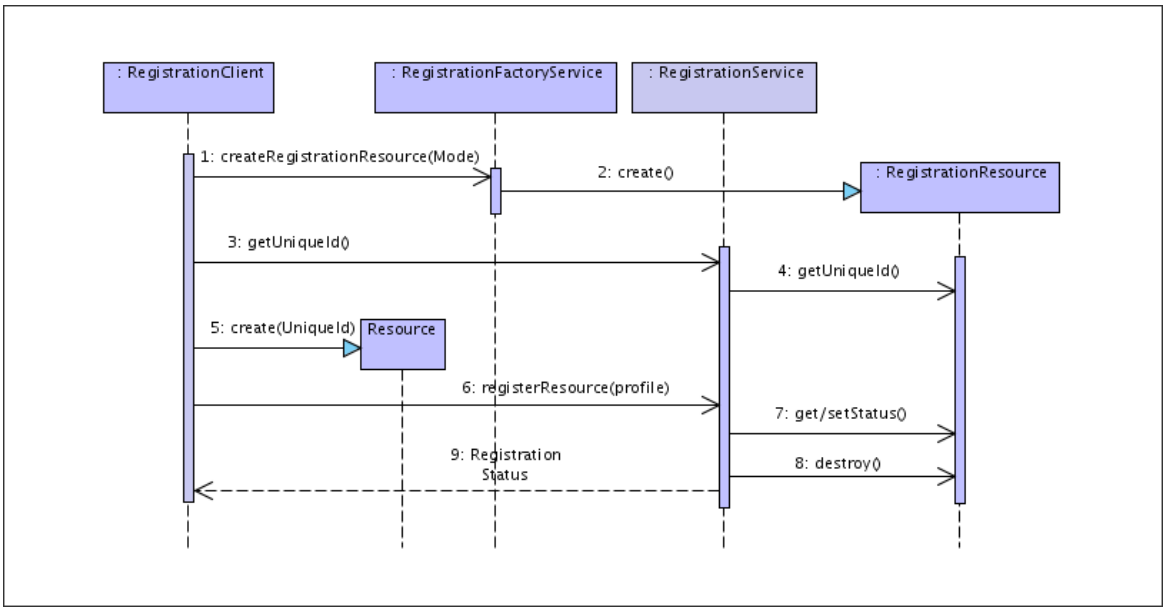

Figure 48. Resource Registration Support - RegistrationResource creation and use

- **registerResource**(Profile) – This operation is invoked to start a new resource registration process. The behavior of this method depends on the *Mode* parameter provided at creation of the RegistrationResource. *Request Mode* and *Automatic Mode* registration processes are shown in Figure 49 and Figure 50 respectively. In the latter the "DL Manager" is really a service that can act as a DL Manager.

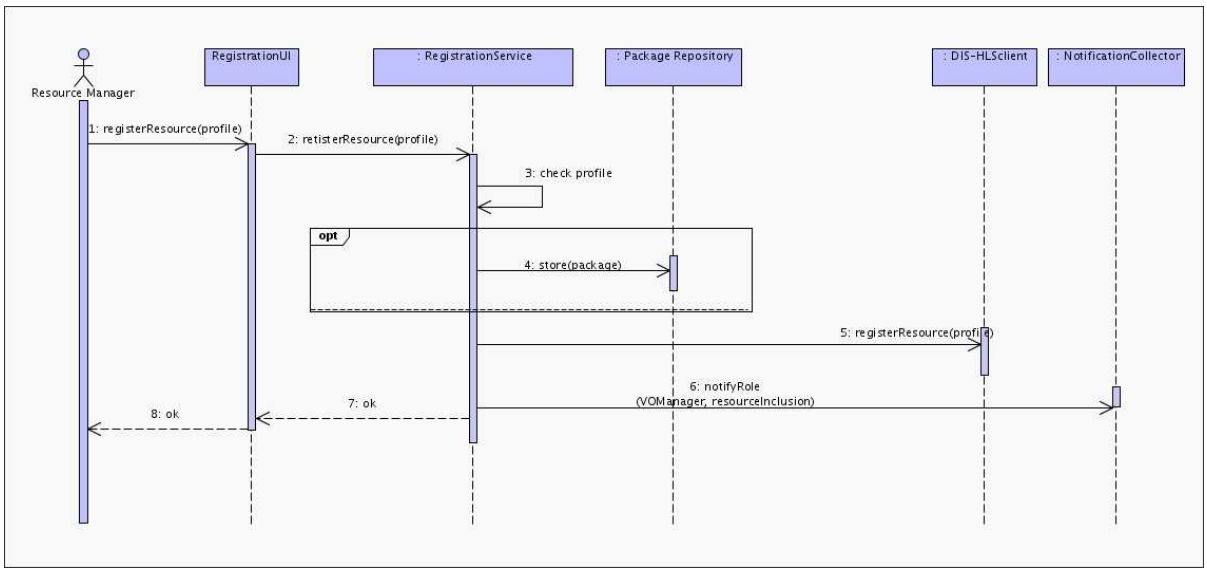

Figure 49. Resource Registration Support - Request Mode Registration Process

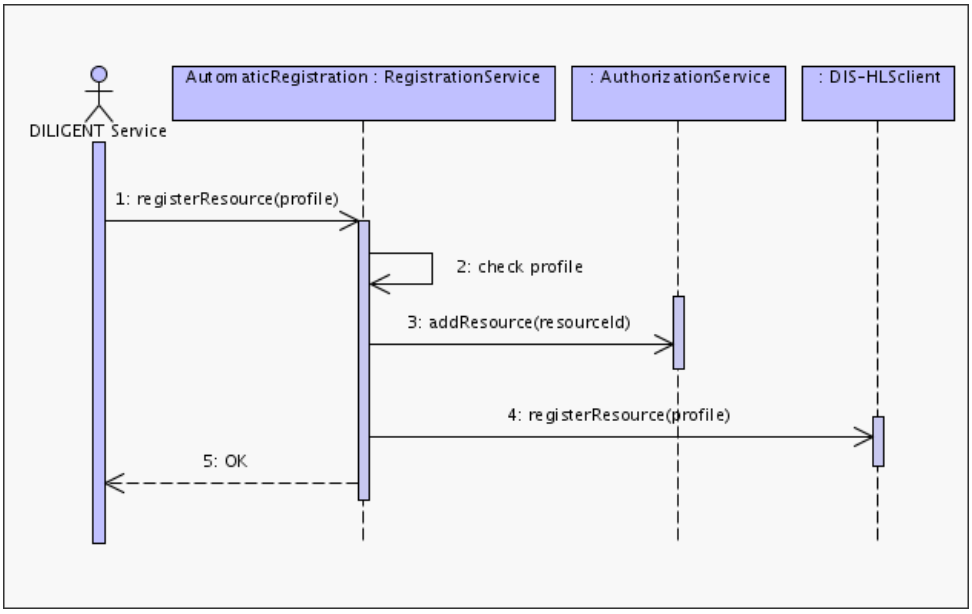

Figure 50. Resource Registration Support - Automatic Mode Registration Process

- **destroy()** – This operation destroy the resource.

7.5.4.2.1.3 Profile description

Only the RegistrationFactoryService profile is registered in the DIS by the local HNM (see Section 6.5.2.3).

[to be detailed in D.1.2.3]

- **RegistrationFactoryServiceSecurityDescriptor1: default**
  - **Authorization Chain**: only VO-level authorization is performed through a handler using *Authorization API* provided by DVOS.
    - **Authorization Criteria**: Only *Resource Managers* can invoke operations of this service.
  - **CreateRegistrationResource**
    - **Identity Scenario**: Caller
    - **Authentication Methods**:
      - **GSISecureMessage** (Integrity)
- **RegistrationServiceSecurityDescriptor1: default**
  - **Authorization Chain**: only local authorization is performed at WS-Resource level; the *identity* authorization handler provided by Java WS Core is used.
    - **Authorization Criteria:** Only the creator of a given RegistrationResource is entitled to use it.
  - **Default Settings**
    - **Default Identity Scenario**: Service
    - **Default Authentication Methods**:
      - **GSISecureMessage** (Integrity)
-

7.5.4.2.1.4 Status Description

[to be provided in D1.2.3]

7.5.4.2.1.5 Dependencies & Requirements

This service relies on the following entities:

DILIGENT services

- DVOS: NotificationService, AuthorizationService
- DIS: DIS-HLSClient
- KEEPER: PackageRepository

7.5.4.2.2 UnregistrationService

7.5.4.2.2.1 State description

The pattern adopted for the *UnregistrationService* design is the WS-Resource Factory Pattern. When the unregistration of a DILIGENT resource is required, the client asks the *UnregistrationFactoryService* to create a new *UnregistrationResource*. Through this *UnregistrationResource*, the client can start and monitor the unregistration process. The client must destroy the *UnregistrationResource* explicitly, before of after the end of the unregistration process. The internal structure of *UnregistrationService* is shown in

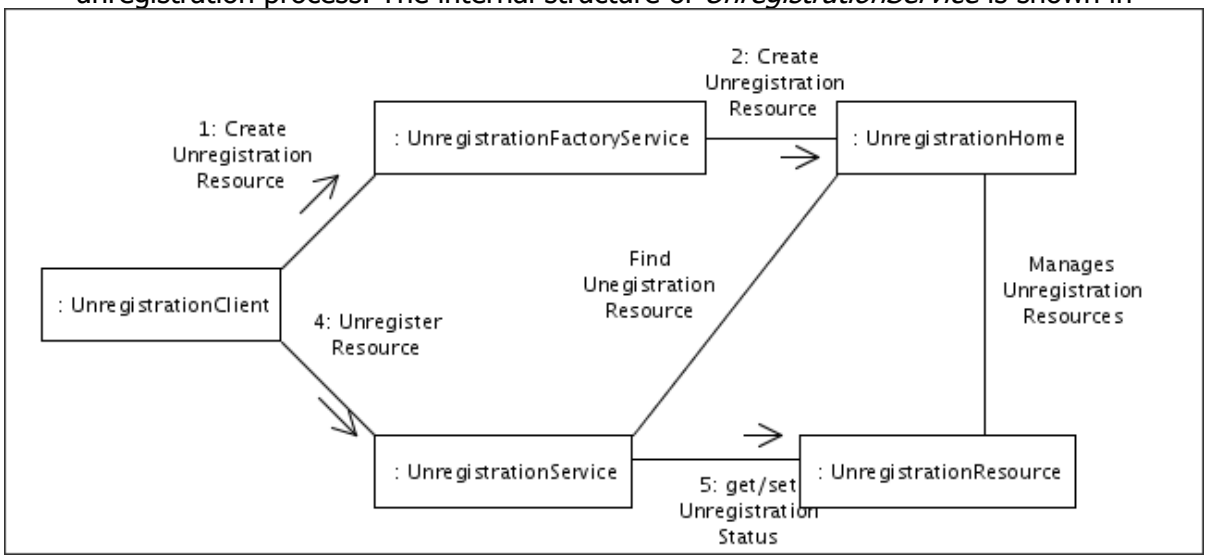

Figure 51. Resource Registration Support - UnregistrationService Structure

.

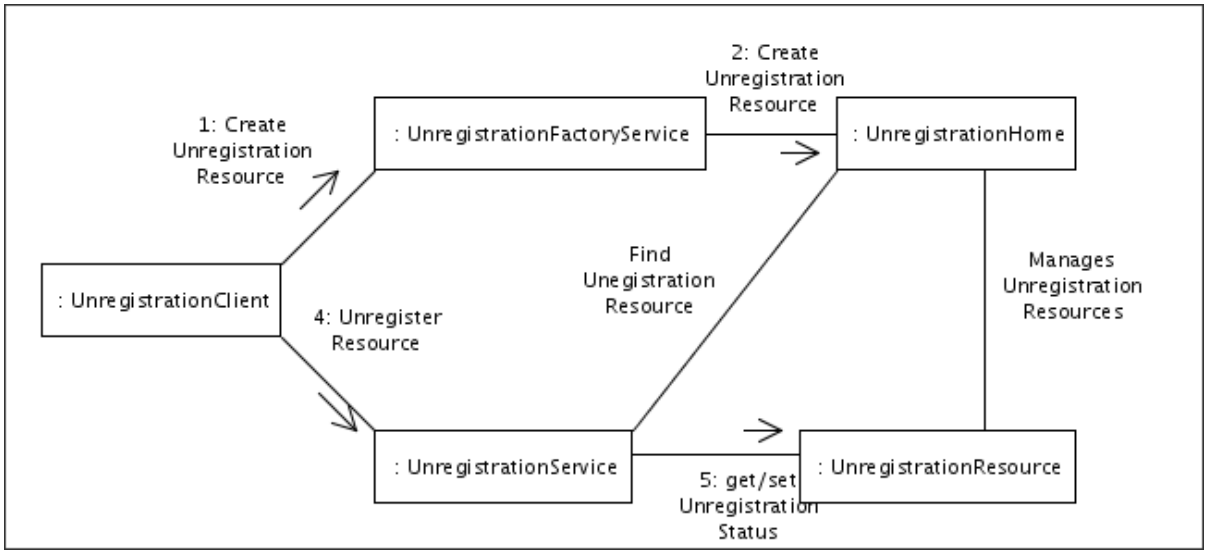

Figure 51. Resource Registration Support - UnregistrationService Structure

7.5.4.2.2.2 Operations

In this paragraph operations are listed together with the service they belong to. This is achieved by placing the name of the service or that of the resource before the signature of the operation.

- **createUnregistrationResource**(Mode) – This operation allows clients to create a new WS-Resource of type *UnregistrationResource*. Such a resource can be used to maintain the state of an unregistration operation. The *Mode* parameter discriminates between the creation of an *UnregistrationResource* operating in *Request Mode* or in *Automatic Mode*. Figure 52 provides an example of *UnregistrationResource* usage.

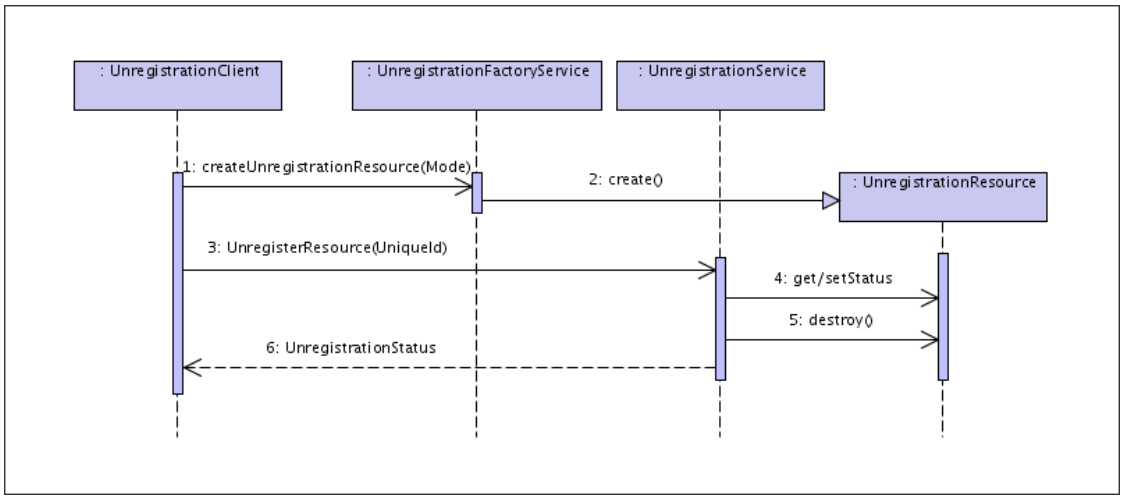

Figure 52. Resource Registration Support - UnregistrationResource creation and use

- **unregisterResource**(UUID) - This operation is invoked to start a new resource unregistration process. The behaviour of this method depends on the *Mode* parameter provided during creation of the *UnregistrationResource*. *UnregistrationResources* operating in *Automatic Mode* can be used to unregister more than one DILIGENT resource. *Request Mode* and *Automatic Mode* registration processes are shown in Figure 53 and Figure 54 respectively.

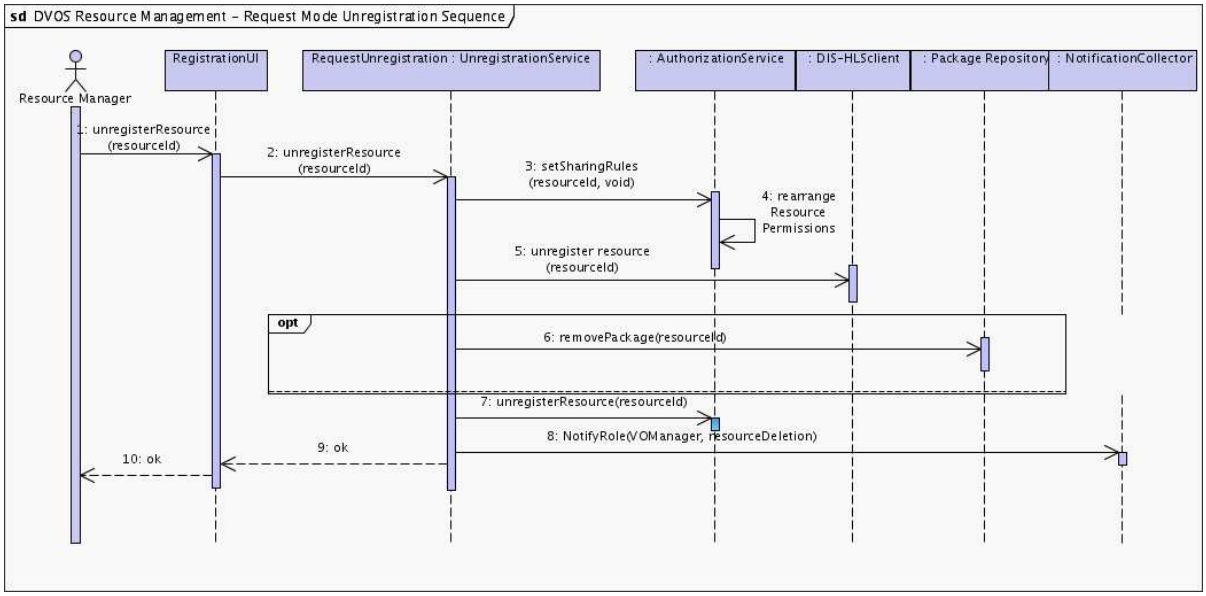

Figure 53. Resource Registration Support - Request Mode Unregistration Process

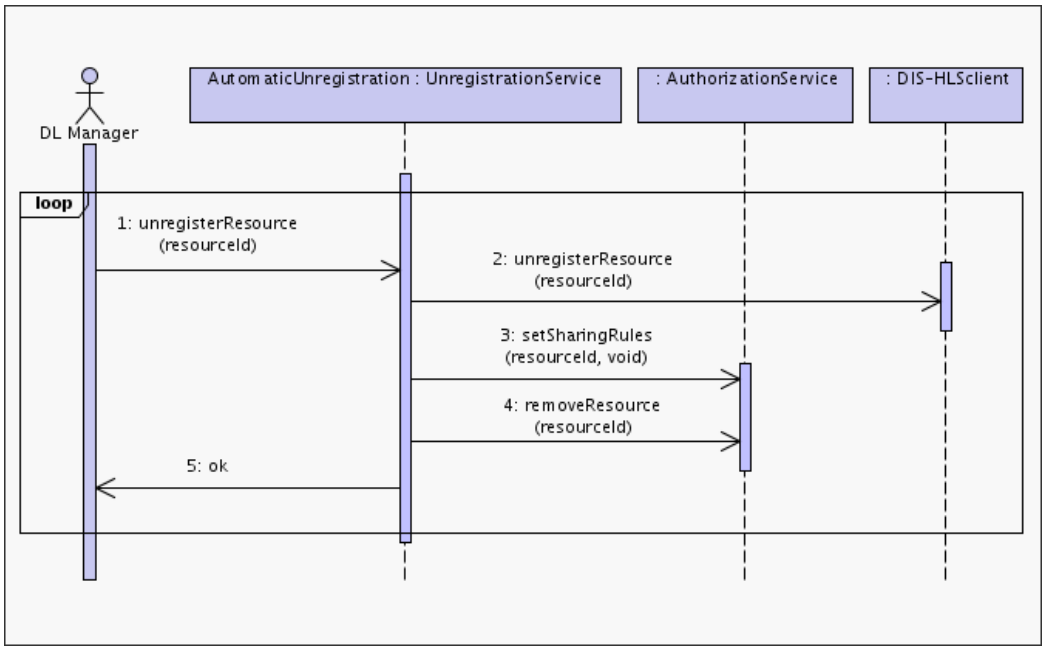

Figure 54. Resource Registration Support - Automatic Mode Unregistration Process

- **destroy**() – This operation destroy the resource.

7.5.4.2.2.3 Profile description

UnregistrationFactoryService only is registered in the DIS.
[to be detailed in D.1.2.3]

- **UnregistrationFactoryServiceSecurityDescriptor1: default**
  - **Authorization Chain**: only VO-level authorization is performed through a handler using *Authorization API* provided by DVOS.
    - **Authorization Criteria:** Only *Resource Managers* can invoke operations of this service.
  - **CreateUnregistrationResource**

- **Identity Scenario**: Caller
- **Authentication Methods**:
  - **GSISecureMessage** (Integrity)

- **UnregistrationServiceSecurityDescriptor1: default**
  - **Authorization Chain**: only local authorization is performed at WS-Resource level; *identity* authorization handler provided by Java WS Core is used.
    - **Authorization Criteria:** Only the creator of a given UnregistrationResource is entitled to use it.
  - **Default Settings**
    - **Default Identity Scenario**: Service
    - **Default Authentication Methods**:
      - **GSISecureMessage** (Integrity)

7.5.4.2.2.4 Status Description

[to be provided in D1.2.3]

7.5.4.2.2.5 Dependencies & Requirements

This service relies on the following entities:

DILIGENT services

- DVOS: NotificationService, AuthorizationService
- DIS: DIS-HLSClient
- Keeper: PackageRepository

7.5.4.2.3 SharingRulesUI

Portlet component.

[to be provided in D1.2.3]

7.5.4.2.3.1 Operations

[to be provided in D1.2.3]

7.5.4.2.3.2 Interaction with other components

[to be provided in D1.2.3]

7.5.4.2.4 RegistrationUI

Portlet component

[to be provided in D1.2.3]

7.5.4.2.4.1 Operations

[to be provided in D1.2.3]

7.5.4.2.4.2 Interaction with other components

[to be provided in D1.2.3]

7.5.4.3 Deployment scenario(s)

At most one instance of the *RegistrationService* may be deployed on each DILIGENT node in order to assure uniqueness of the UniqueId generated during the registration processes. Different instances of the *RegistrationService* and *UnregistrationService* can be deployed independently on different DILIGENT nodes. The Universal Unique IDentifier standard itself assure the global uniqueness of resource identifiers generated from different instances running on different DILIGENT nodes.

8 VDL GENERATOR SERVICE

8.1 Introduction

The VDL Generator Service is the service that enables users/communities to create their own DLs. It allows to define a set of criteria that specify the expected characteristics of the new DL; starting from them, it identifies the set of services required to provide the requested features. In particular, the VDL Generator Service:

- Supports users/communities in specifying the criteria that characterize the new DL, e.g. the information space, the required functionalities, the characteristics of these functionalities;
- Selects the appropriate pool of services and information sources required to implement a DL that satisfies the specified criteria;
- Notifies to the DL Management service (see Section 6.5.2.1) the identified services.[31]

This service is also characterized by a high interaction with a user via a graphical user interface. When specifying the architecture of such kind of interactive service, the challenge is to keep the functional core independent from the user interface. In fact, while the core is based on the functional requirements and usually remains stable, the user interface is often subject to changes and adaptation. This base design choice has consequences on the entire design of the VDL Generator Service. In the following paragraphs we introduce the system functionalities covered by this service and the different architectural views needed to model the VDL Generator Service specification.

8.2 Use-Case View

By analyzing the deliverable D1.1.1 "Test-bed Functional Specification" [1] a set of functionalities has been identified as related with the VDL Generator Service. In particular, the following functionalities, described as UCs in D1.1.1, are provided directly from the service:

4.2 DLs Management

4.2.1 Define a DL

4.2.2 Select Archives

4.2.3 Select Services

4.2.4 Define Configuration

4.2.5 Define Web Portal Configuration

4.2.6 Ask for DL Creation

4.2.7 Modify a DL

4.2.8 Ask for DL Update

4.2.9 Dispose a DL

4.2.10 Preserve content

4.2.11 Ask for DL Removal

In order to do its tasks, the VDL Generator service relies upon the following functionalities that must be provided by other DILIGENT services:

4.3.18 Browse Available Resources

[31] From the DoW.

6 Index&Search Management
6.6.2 Search Archive
6.6.2 Search for Object/Resource

4.5 Users Management
4.5.26 Invite a Group to a DL
4.5.21 Invite a User to a DL

4.6 Notifications management
4.6.2 Notify Role

These functionalities are rearranged into packages and use cases representing the service functionalities in Figure 55. The same diagram also reports the dependencies and relationships with other packages.

The following rationale has driven the creation of the Use Case view and the whole design of the service:

a. use cases having the same name of those reported in the list represent the same functionality,
b. the functionalities related to the selection of resources, their configuration and the "ask" operations (Ask for DL Creation, Ask for DL Update, and Ask for DL Removal) are covered by both the VDLGeneratorUI and the VDLGeneratorLogic packages,
c. other UCs do not come directly from any user functionality but they are needed to support the previous ones, e.g. Query KB.
d. The single "DL" concept used in D1.1.1 has been split here in two concepts:
   - VDL – we use the term "VDL", which stands for Virtual Digital Library, when we refer to the definition phase of a DL; in these early stages a VDL is just a set of definition criteria that describe the DL that is subsequently generated starting from them. These criteria are both those provided by the DL Designer and those elaborated by the VDL Generator service;
   - DL – the term DL is used starting from the generation phase of a DL; such phase, performed by the DL Management functionalities of the Keeper service (see section 6.2), makes concrete the VDL definition criteria in a real DL.

The role of the VDL Generator Service is to produce and store VDL Definitions that the Keeper service (its DL Management component) can use to set up a real DL. A VDL Definition is composed by set of VDL Definition criteria, including:

1. the definition as expressed by the DL Designer
2. the definition as computed by the VDL Generator service (set of configured services/users/resources)

Each UC is described in the following subsections.

VDLGeneratorUI

This package contains functionality representing the graphical user interface functions enabling the DL Designer to interact with the VDL Generator Service.

- **Define a VDL -** This functionality represents the activity the DL Designer performs by means of the graphical user interface in order to specify the characteristics of the

DL (named *VDL definition criteria*). This activity, supported by the VDLGeneratorLogic package, is capable to prevent the DL Designer to make inconsistent choices. For instance, the system prevents the user to chose services that are not able to work together like a search service that, roughly speaking, is not capable to interact with the selected information source (we have a DL that is not able to discover the documents it contain!). This functionality also allows to specify users that are entitled to have access to the DL that is generated starting from the VDL Definition by using the Select User/Group functionality offered by User and Group Management presented in Section 7.4.

- **Select Resources -** This functionality, supported by the VDLGenerator package, is in charge to propose, step-by-step, the pool of resources that can be used to define the VDL. For instance, the DL Designer is enabled to select the archives the real DL will be equipped with after having identified them by a search, and to select the search services the DL can be equipped with in accordance with the until now VDL definition choices performed.
- **Modify a VDL -** This functionality represents the activity that the DL Designer performs in order to update/review the characteristics of a previously defined VDL. Then, these updates are reported in the real DL in terms of services and/or packages modification.
- **Dispose a VDL -** This functionality represents the activity the DL Designer performs in order to remove a previously defined VDL. This action enables also the DL Designer to decide about preservation policies to apply to DL Resources via the Establish Preservation Action use case.
- **Establish Preservation Action -** This functionality represents the activity that allows the DL Designer to establish the actions to perform in order to preserve the status of DL Resources before to remove them.

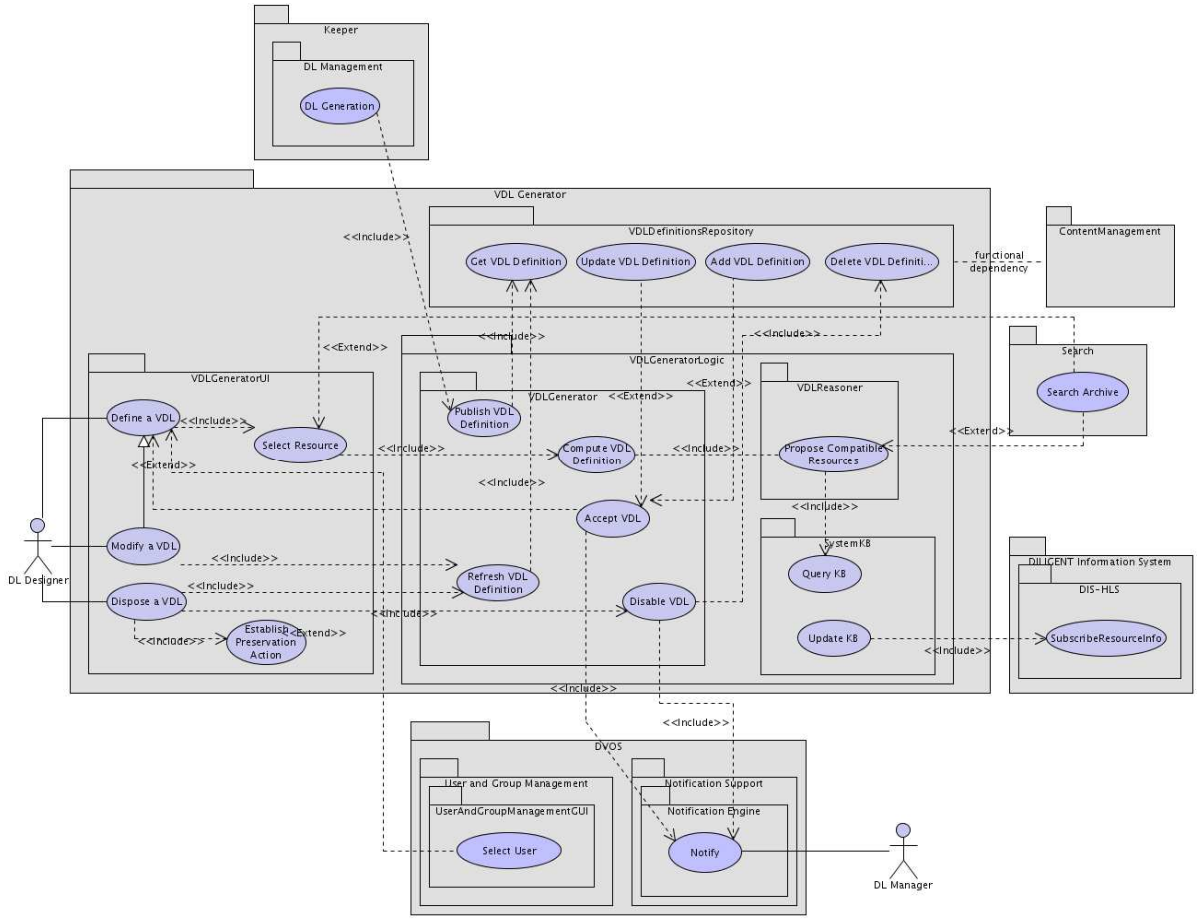

Figure 55. VDL Generator - Use-Case View

VDLGenerator

This package contains functionalities enabling the VDL Generator Service to accept and perform the actions required by the DL Designer via the GUI.

- **Compute VDL Definition** - This functionality represents one of the main activities carried out by the VDL Generator Service, i.e., the computation of the VDL definition in accordance with the choices performed by the DL Designer by means of the "Define a VDL" functionality. It is supported by the Propose Compatible Resources function described later to derive the pool of resources that must be included into the VDL Definition.
- **Accept VDL** - This functionality represents the action the DL Designer performs in order to make the definition of the VDL persistent and to begin the procedure to automatically create the real DL. The first task is performed by adding (or updating, if the functionality is invoked in the context of "Modify a VDL" functionality) the new VDL Definition to the Definitions repository. The second task consists in notifying the DL Manager about the request performed by the DL Designer to create (or update) the DL.
- **Refresh VDL Definition** - This functionality represents the activity performed by the VDL Generator Service in order to recall in line the VDL Definition. In fact, as presented later, the VDL Generator Service maintains in line the model of a DL in order to support the user interface in performing its tasks. This functionality is thus invoked each time a previously defined VDL must be modified or removed by the DL Designer in order to allow the VDLGeneratorUI functionality to render the VDL Definition as stored within the Definitions repository.

- **Disable VDL** - This functionality processes the request of a DL Designer to remove one of the VDLs previously defined. It is composed by two sub tasks, the invocation of the functionality entitled to remove the VDL Definition from the Definitions repository and the notification to the DL Manager about the request, to remove the real DL generated from that Definition. This operation is delayed in order to guarantee a safely termination of the open accessing connections.
- **Publish VDL Definition** - This functionality represents the way by which other services (i.e. the Keeper) can access the computed list of resources forming the a defined VDL. This list contains the resources identified as well as their configuration parameters (in case of resources that must be deployed). Moreover the Definition also contains the list of users to be enabled to have access to the future DL (in the case of restricted access).

VDLReasoner

This package contains the functionality representing the algorithm in charge to identify the resources needed to fulfil the DL Designer requirements.

- **Propose Compatible Resources** - This functionality represents the algorithm that is in charge to automatically identify the pool of resources to form a VDL starting from the characteristics provided by the DL Designer. From a functional point of view, this functionality is invoked each time a new VDL requirement is expressed by the DL Designer and thus it is needed to re-compute the pool of resources capable to satisfy it.

SystemKB

This package contains the functionality representing the population and the querying of the knowledge base maintained by the VDL Generator about the resources available.

- **Query KB** - This functionality represents the action of querying the knowledge base of the system performed by the previously described reasoning algorithm in order to perform its tasks. The interaction mechanism is completely hidden to the other DILIGENT services as well as the query language and the tool used to maintain the knowledge base.
- **Update KB** - This functionality represents the action of populating on start-up and then updating of the knowledge base, as well as its initial population with the information about the resources belonging to DILIGENT needed to support the Propose Compatible Resources functionality.

[to be detailed in D1.2.3]

VDLDefinitionsRepository

This package contains functionality needed to model the operations for storing, retrieving and removing VDL Definitions. VDL Definitions are stored as files distributed on the storage elements available on the Grid infrastructure.

- **Get VDL Definition** - This functionality represents the retrieval of a previously stored VDL Definition from the Definitions database.
- **Add VDL Definition** -This functionality represents the adjunction of a new VDL Definition to the Definitions database.
- **Update VDL Definition** - This functionality represents the updating of a previously stored VDL Definition to the Definitions database. The Definitions database is really a

mapping table between the DL identifier and the file/files containing the relative Definitions, thus this operation must be completed updating all the files accordingly.

- **Delete VDL Definition** - This functionality represents the removal of a previously stored VDL Definition to the Definitions database. This operation must remove all the files constituting the VDL Definition that are distributed somewhere in the Grid infrastructure.

8.3 Logical View

In order to design the VDL Generator Service we decided to organize its logical structure following the *Model-View-Controller* architectural pattern [2]. This pattern divides an interactive application into three main components:

- a model (the *VDLGeneratorModel* class) that contains the core functionality and data;
- some Views (the *VDLGeneratorView* class) that display information to the user;
- some Controllers (the *VDLGeneratorController* class) that handle user input.

A further major class is the *VDLDefinitionRepository* that models the storage of the VDL Definitions. All these classes and their relationships are shown in Figure 56 and briefly described in the following subsections. Classes and packages of the Logical View are related with the Use Case View in the following way: packages having the same name represent the same functional area.

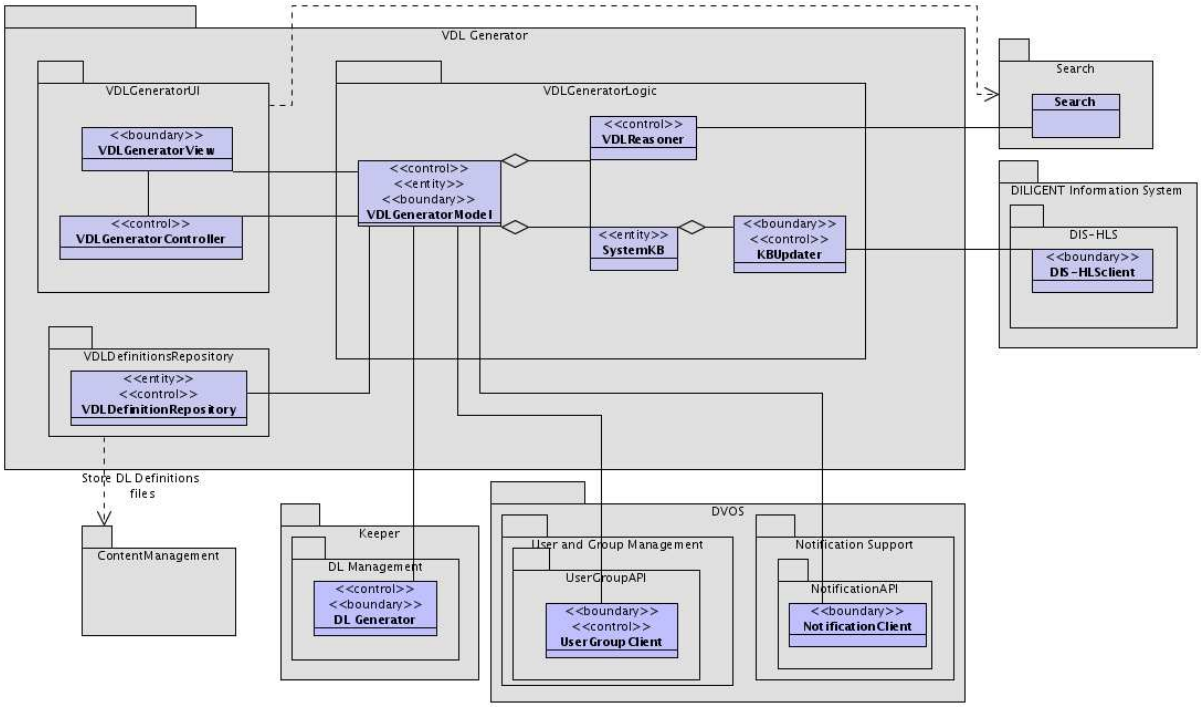

Figure 56. VDL Generator - Logical View

VDLGeneratorUI package

This package represents the User Interface of the VDL Generator Service. It is composed of two classes, *VDLGeneratorView* and *VDLGeneratorController*. The former is in charge of displaying information to the user while the latter is responsible for managing user inputs received by the view component and translating them into service requests for the *VDLGeneratorModel* (described in the following section) or the view itself.

VDLGeneratorLogic package

This package represents the functional core of the service. It encapsulates the appropriate data needed to support the service as well as the procedures/methods needed to perform the service functionalities.

It is composed of

- A V*DLGeneratorModel*, that represents the main components of this package. It is the front-end to all the functionality and data covered by this package;
- A *VDLReasoner*, that represents the components in charge to automatically identify the DL components/resources needed to fulfil the user requirements; the algorithm, starting from the knowledge stored in the SystemKB, is capable to select and assemble the pool of available components in a suitable way to fulfil the VDL requirements. The algorithm performs a quite complicated job because there are potentially thousand of choices and constraint to obtain a well-suited solution.
- A *SystemKB*, that represents the knowledge base needed to support the other components. This knowledge base maintains all the needed information about the available components/resources of the DILIGENT system;
- The *KBUpdater*, that is in charge to maintain the data stored in the SystemKB in line with the DILIGENT status (i.e., the available resources and their related information) available via the DILIGENT Information System;

Details about the DLReasoner as well as the SystemKB will be supplied in further Sections. The problem that these classes plan to solve is related to the automatic identification of the most suitable set of DLComponents needed to fulfil the DL Definition. The solution proposed is to express both the VDL Definition criteria and the DLComponents together with the related applicability/usability constraint by using a knowledge representation mechanism. Then by adopting the inference mechanism the system will be able to identify the set of DLComponents needed to satisfy the VDL Definition criteria.

VDLDefinitionsRepository

This package models the repository in charge to maintain the Definitions of the various DILIGENT DLs. The repository must be capable to handle the VDL Definitions as originally expressed by the user and their rearrangement performed by the VDLReasoner in order to express the VDL in terms of its forming components. It:

- Supports the "Modify a VDL" functionality allowing the VDLGeneratorUI to show to the DL Designer the choices performed in the previous definition phase and thus to rearrange them;
- Supports the DL Creation phase performed by the Keeper by supplying the DL definition expressed in terms of DLComponents (and related configuration parameters) and DL users.

8.4 Deployment View

The deployment of the VDL Generator Service, as shown in Figure 57, is based on a three-layered configuration:

- The *front-end* is mainly formed by the components that provide the presentation layer of the service, i.e. the VDLGeneratorView and the VDLGeneratorController. These components implement the logical classes having the same name;
- The *middle-tier* is composed by the logic of the VDL Generator Service, i.e. the VDLGeneratorModel, the SystemKB and the VDLReasoner. For performance reasons

the latter two components are deployed on the same node where the VDLGeneratorModel service is deployed;

- The *back-end* is the storage component in charge to maintain the VDL Definitions.

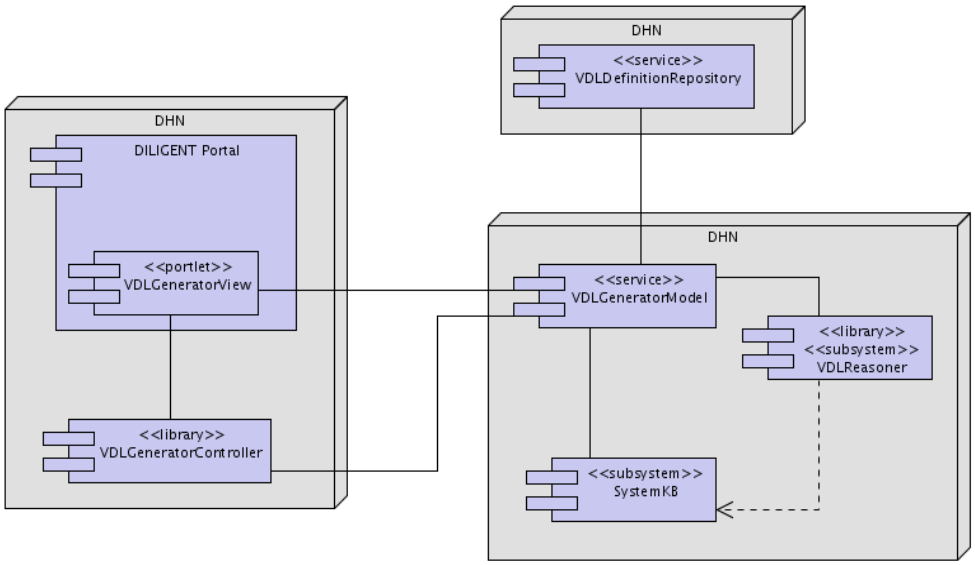

Figure 57. VDL Generator - Deployment View

8.5 Service design

8.5.1 Design considerations

The VDL can be considered a centralized component w.r.t. the entire infrastructure. It is used only by the DL Designer to create new VDL Definitions. Thus the complexity of the service is concentrated in the logical layer that has to elaborate the DL Designer requests and produces solutions that can be managed by the Keeper service.

The difficulties encountered within the logic of the application are related to the identification of an appropriate knowledge representation mechanism/formalism. It must be powerful enough to allow to express the knowledge we need to represent as well as it must have an inference mechanism that is computable in an human acceptable time. Moreover, it must exist an system that provides support for the implementation of the identified mechanism, in DILIGENT we do not have the effort and the goal to implement a knowledge representation system. In our best knowledge, a suitable candidate solution is the Datalog language, and in particular the disjunctive Datalog extension. Datalog is a rule-based *declarative* language. This means that a user has not to write a program that solves some problem but instead specifies what the solution should look like, and a Datalog inference engine tries to find the way to solve the problem and the solution itself with the available registered components. This is done with *rules* and *facts*. Facts are the input data, and rules can be used to derive more facts, and the solution of a given problem. Disjunctive Datalog is an extension of Datalog in which the logical OR expression (the disjunction) is allowed to appear in the head of a rule. Details on this system are provided in the VDLReasoner description.

Another important issue is also the presentation layer for the aspect of the usability of the service. The resulting GUI must be *easy to use* as well as *adaptable to different kind of users* since usually the DL Designer is a user without technical skills. Supported by the

logical layer, the GUI must drive the user in specifying the VDL Definition criteria avoiding to express incompatible needs.

8.5.2 Components

We have already introduced the VDL Generator components in the deployment view section (8.4). To better understand how these components interact each other, we present in Figure 58 a sequence diagram showing the interactions among them during the definition of a new Virtual Digital Library.

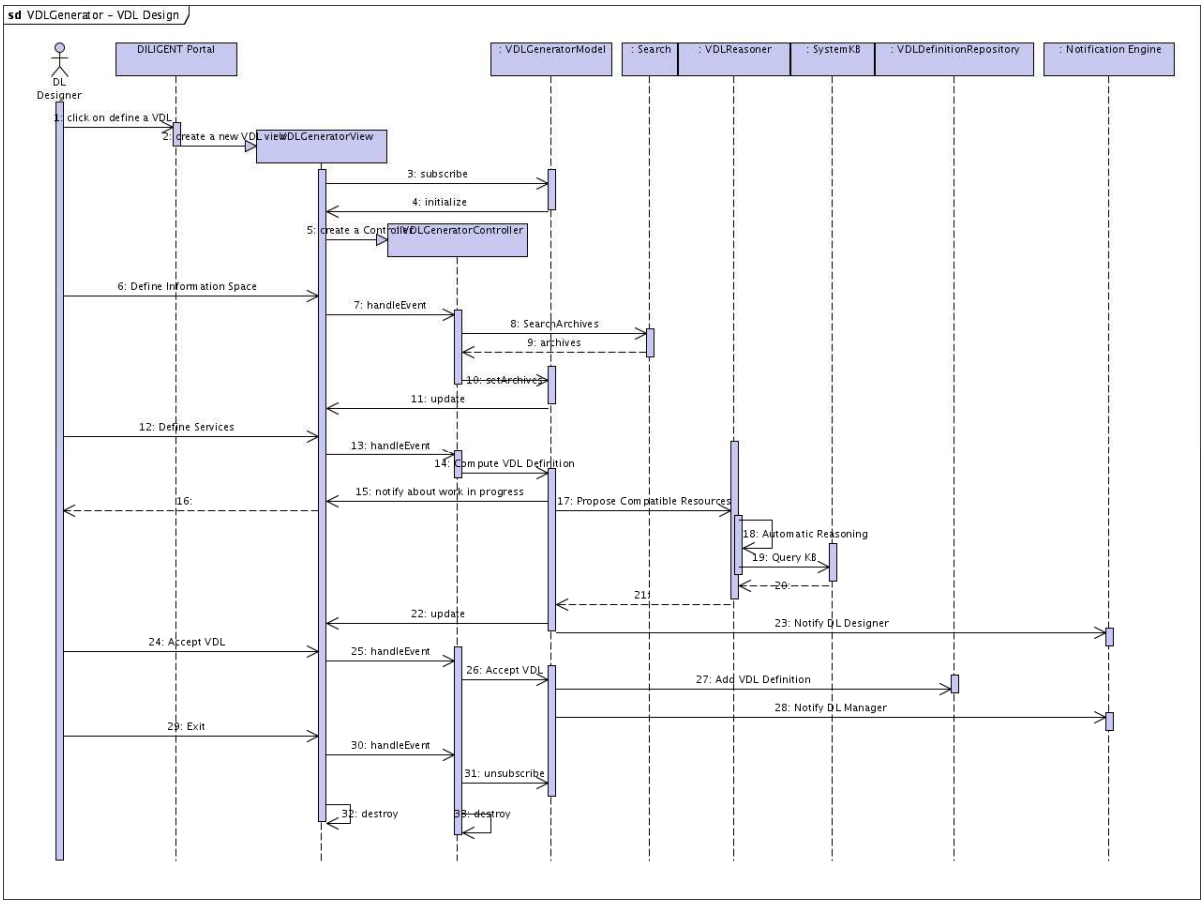

Figure 58. Define a VDL

The VDL definition process is organized into the following steps:

1. The DL Designer, after having logged in into the system, is entitled to perform the "define a VDL" by clicking on the appropriate link offered by the DILIGENT Portal;
2. The Portal creates a new VDL view by instantiating a VDLGeneratorView portlet;
3. The VDLGeneratorView subscribes itself to the VDLGeneratorModel Service that is up and running.
4. The VDLGeneratorModel Service initializes the portlet by supplying it a set of data to use for drawing and filling in the user interface. Among these data there are parameters supported by the VDLGeneratorModel to define the characteristics of the VDL. For instance, there is the list of archives (used to populate a check list) that can be selected by the DL Designer as well as the supported exploitable features to express requirements about services and other non-functional needs. Then, the graphical user interface is built according to these parameters;

5. The VDLGeneratorView creates its controller instance which is in charge to manage all the input events;
6. By interacting with the VDLGeneratorView portlet the DL Designer is enabled to specify the characteristics of the Information Space the VDL he/she is requiring. To support this functionality, the user interface supplies two mechanisms:
    - The DL Designer browses the list of available archives and selects those the DL Community is interested in; clearly, this list is a snapshot of the currently available archives. If new archives will become available in DILIGENT, she/he will be notified and by performing the Modify a VDL Use Case she/he can add the new archives to the VDL.
    - The DL Designer expresses a characterization criterion that can be passed to the DILIGENT Search service[32] (this service is described in D1.4.1 Index & Search services specification interim report) to identify the list of archives fulfilling the specified requirement. The kinds of criteria that are allowed to be expressed strongly depend by the available Search service instance and by its query language. For instance, the ARTE community expresses as requirement the possibility to identify the archives just by supplying a picture and retrieving all those that contain similar images.
7. In case the DL Designer expresses a criterion that needs to invoke the Search service the following steps (8 and 9) are performed, otherwise the VDLGeneratorController skips them and executes step 10;
8. A query specifying the characterization criterion is sent to the DILIGENT Search instance;
9. The list of the Archives fulfilling the query is returned to the VDLGeneratorController;
10. The list of archives, identified both with selection or search modalities, is passed to the VDLGeneratorModel;
11. The VDLGeneratorModel updates its internal partial representation of the VDL and the notifies the VDLGeneratorView portlet to update the user interface by taking into account the until now performed choices;
12. By interacting with the VDLGeneratorView portlet the DL Designer is enabled to specify the characteristics of the Services the VDL he/she is requiring for. Once again the user interface supplies an intuitive mechanism to express the characteristics of the VDL. These characteristics can be specified by using a scroll list of allowed values to be defined in the following phases of the Service design (D1.2.3). However, they strongly depend by the characteristics that each service advertises and exposes;
13. Once the DL Designer ends the definition phase, the VDLGeneratorController is called in order to handle appropriately the inputs;
14. The VDLGeneratorController calls the VDLGeneratorModel in order to validate, complete and verify the VDL definition;

[32] We assume here that, at VDL Definition time, an instance of a Search service is deployed and available within the DILIGENT infrastructure. More precisely, we assume that a DL with a set of service instances is available to support the activity performed via the DILIGENT portal. It is important to highlight here that this DL is not defined and created with the support of the VDL generator and of the Keeper. This pre-existing DL is created and configured manually because it pre-exists to the services needed to create DLs.

15. In order to avoid the DL Designer to wait on line while the reasoning algorithm execute its tasks, the VDLGeneratorModel notifies the VDLGeneratorView to show a message explaining the user about the process in progress;
16. The VDLGeneratorView portlet displays the appropriate message enabling the DL Designer to logout from the portlet. At the end of the reasoning phase, i.e. when the VDL Definition is computed by the system, the DL Designer is notified in order to enabling he/she to accept or reject the proposed DL;
17. Without human intervention, the VDLGeneratorModel, with the support of the VDLReasoner and the SystemKB subsystems, finalizes the definition criteria;
18. The VDLReasoner starts an iterative reasoning process (presented in the next sections) whose goal is to verify the correctness of the criteria of the VDL and to complete them by identifying all the components needed to build a real DL compliant with the specified criteria;
19. To perform its reasoning algorithm the VDLReasoner needs to query the SystemKB that supplies information of the available components, mainly their dependencies;
20. The SystemKB replies to the query following the semantic model underlying the entire decision process. This model is reported in Section 8.5.2.1.1.1;
21. At the end of the reasoning process the VDLReasoner notifies the VDLGeneratorModel with the Definition of the VDL computed;
22. The VDLGeneratorModel updates its internal representation of the VDL and then notifies the VDLGeneratorView;
23. The VDLGeneratorModel notifies the DL Designer about the end of the computational phase and the availability of the DL Definition as reviewed by the system identifying the DL Components and their interoperability;
24. The VDLGeneratorView updates the user interface with the complete definition of the DL and if it is acceptable, the DL Designer accepts this definition by clicking on the "Accept DL" button;
25. The VDLGeneratorController is called in order to handle this definition finalize event;
26. The VDLGeneratorController calls the VDLGeneratorModel asking to manage the DL definition;
27. The VDLGeneratorModel calls the VDLDefinitionsRepository in order to store the DL as defined by the user as well as the computed definition to supply to the Keeper;
28. The VDLGeneratorModel calls the NotificationEngine in order to notify the DL Manager about the new DL to create starting from the computed VDL Definition;
29. The user finished the definition phase by clicking on the exit link of the VDLGeneratorView portlet;
30. The VDLGeneratorView notifies the VDLGeneratorController about the exit action performed by the user and prepares itself to be deallocated;
31. The VDLGeneratorController sends an unsubscribe request to the VDLGeneratorModel in order to be removed from the list of subscribers;
32. The VDLGeneratorView is removed;
33. The VDLGeneratorController is removed.

8.5.2.1 VDLGeneratorModel

The VDLGeneratorModel service represents the access point to the VDLGeneratorLogic functionalities. As a consequence it is the service in charge to offer the core functionalities

of the VDL Generator Service. It is implemented as a WSRF service and we assume that an instance of this service is always available within the DILIGENT infrastructure.

8.5.2.1.1 The SystemKB and the DLReasoner

8.5.2.1.1.1 The Reasoning

The VDL Generator Service, and in particular the functionalities related to the definition of a VDL and the automatic identification of the pool of resources needed to fulfil the VDL requirements, can be modelled as a configuration problem.

The following two aspects characterize these kinds of problems:

- The artifact to build is composed by instances of components;
- Components interact in predefined ways, i.e. their dependencies from other components are well-known.

All the systems for product configuration converge on describing this problem representing two kinds of knowledge: i) the *domain description*, i.e. a description of all the types of components available, and ii) the *specification of the desired product*. This is also the approach that we plan to adopt.

In particular, for domain modeling we adopt the component-port approach. As reported in Section 3, DL components are characterized by three elements: the type, the attributes, and the ports:

- *Types* allow organizing components into a hierarchy that can be used during configuration.
- *Attributes* specify descriptive features, such as functional or technical characteristics, configuration parameters, etc. Each attribute has a single value or can take values from a predefined range.
- *Ports* are used to establish connections between components. Usually when defining ports restrictions may be imposed on the type and number of components that can be connected to it. Constraints placed on ports are the natural way to express compatibility between components. They allow expressing conditions on attributes and ports that must hold in the model built to satisfy the DL requirements.

All these characteristics will be carefully identified, defined and used to annotate all the DILIGENT DL Components in the next design stages. We will define the rules and the mechanisms by which these characteristics can be specified. Then, it is up to each service to use this mechanism in the most appropriate way needed to specify the characteristics of its service.

In the following we present a brief example that is intended to help understanding the usage of this knowledge representation mechanism.

An example

Suppose that DILIGENT is equipped with:

- A Collection of documents whose descriptive metadata are available in DublinCore format [13];
- An Index service capable to be configured to support any metadata format;
- A Search service capable to be configured to support any metadata format;

Suppose now that the DL Designer expresses the requirement to have a DL whose metadata format is DublinCore and that one of the characteristics of each DL is that it is made by Collections.

The VDLReasoner, by adopting a general constraint like "if a DL has a metadata format X then collection has metadata format X" is capable to identify the former collection because it

declares the attribute Metadata="DublinCore". Then, if there is another general constraint stating that each DL must have a search service and that the supported metadata format is the same of that of the DL, the VDLReasoner is capable to identify the previously described search service. As the search service declares to have a port that is of type Index and must be filled with an Index having the same Metadata format the VDLReasoner must be enabled to select and use the Search if and only if it is capable to identify and assign an index with the desired characteristics to the appropriate port. The VDLReasoner will be probably implemented by exploiting the software release under the DLV project[33] that released a Disjunctive Datalog System.

8.5.2.1.2 State description

The VDLGeneratorModel creates a new WS-Resource for each VDL Definition process. The WS-Resource is created when the VDLGeneratorView contacts the service to send a new subscription request for a new VDL Definition and destroyed when the related unsubscription request is received.

8.5.2.1.3 Operations

- **Subscribe()**
- **SetArchive()**
- **ComputeVDLDefinition()**
- **AcceptVDL()**
- **Unsubscribe()**

[to be detailed in D1.2.3]

8.5.2.1.4 Profile description

[to be provided in D1.2.3]

8.5.2.1.5 Status description

NONE

8.5.2.1.6 Dependencies and Requirements

DILIGENT services

- Search: Search service
- Keeper: DL Management service
- DILIGENT portal

[to be provided in D1.2.3]

gLite services

NONE

8.5.2.2 VDLGeneratorController

The VDLGeneratorController is in charge of handling the inputs provided by the DL Designer via the VDLGeneratorView portlet. This component has been designed as a library and it has to be deployed on the same DHN where the VDLGeneratorView is hosted by the DILIGENT Portal.

Each handled input is appropriately elaborated and translated in an invocation to the VDLGeneratorModel service.

[33] http://www.dbai.tuwien.ac.at/proj/dlv/

8.5.2.2.1 Operations

- **HandleEvent**()

8.5.2.2.2 Dependencies and Requirements

[to be provided in D1.2.3]

8.5.2.3 VDLGeneratorView

The VDLGeneratorVIew is designed as a portlet that allows the DL Designer to create new VDL Definitions.

8.5.2.3.1 Operations

- **CreateNewVDLDefinition()**
- **DefineVDLInformationSpace()**
- **DefineVDLServices()**
- **Accept VDL()**

[to be detailed in D1.2.3]

8.5.2.3.2 Interactions with other components

NONE

8.5.2.4 VDLDefinitionRepository

The VDLDefinitionRepository is a storage manager for the VDL Definitions defined by DL Designers. It is designed as a WSRF service.

8.5.2.4.1 State description

[to be provided in D1.2.3]

8.5.2.4.2 Operations

- **AddVDLDefinition()**
- **RemoveVDLDefinition()**
- **UpdateVDLDefinition()**
- **RemoveVDLDefinition()**
- **GetVDLDefinition()**

[to be provided in D1.2.3]

8.5.2.4.3 Profile description

[to be provided in D1.2.3]

8.5.2.4.4 Status description

[to be provided in D1.2.3]

8.5.2.4.5 Dependencies and Requirements

[to be provided in D1.2.3]

8.5.3 Deployment scenario(s)

The deployment scenario of the described components is quite simple because the VDL Generator can be considered as a centralized component w.r.t. the entire infrastructure. It

is not expected to have a number of replicas of the service for performance reasons since there are no strong requirements in terms of response time.

Since it is the starting point of the entire process of DL generation, an instance of each component is manually deployed when the infrastructure is set up. Other instances of the components can be automatically deployed by the Keeper (e.g. its DL Management service) if one of the previous ones goes down for any reason.

9 CONCLUSION

This report presents the result of the activity conducted within the various design tasks: *T1.2.1.a Information Service design*, *T1.2.2.a Broker & Matchmaker Service design, T1.2.3.a Keeper Service design, T1.2.4.a Dynamic VO Support Service design*, and *T1.2.5.a VDL Generator Service design* of the *WP1.2 DL Creation & Management* of the DILIGENT project during the period February 1st - August 31st 2005.

The *DL Creation & Management services specification report* is given in the formal notation recommended by the Unified Process software engineering methodology, according to Annex I – "Description of Work"; this formal notation is accompanied by texts as expressed by the DILIGENT technological partners.

To improve the readability of the document, preserving in the same time the peculiarities of each described services, this report contains for each technical section two parts.

The first part is common to all services and presents, through a set of UML diagrams, the functional, logical, and deployment view with the aims of illustrating the most significant architectural decisions that have been made, the boundaries of the services, and the interaction with its surrounding.

The second part is service specific and reports main information about the service decomposition in components. In particular, for each service the design considerations identify the issues that will be addressed or resolved in the complete detailed design solution and for each component an in-depth analysis is explained through the state, operations, profile, status, dependencies, and requirements description.

Finally, it is important to remark that this service specification complete the D1.2.1 DL Creation & Management Services Specification report maintaining its strongly relationship with the *D1.1.1 Test-bed functional specification*. This means that using this report became now feasible to realize how the functionalities related to the user requirements, reported in D2.1.1 *ARTE Scenario Requirements Analysis Report* and *D2.2.1 ImpECt Scenario Requirements Analysis Report*, are served by service components of the DILIGENT infrastructure.

Appendix A. Java WS Core Security

This appendix aims to provide some guidelines in designing DILIGENT service security. This is not an overview of the security framework of by the Java WS Core, for such an overview sees http://www.globus.org/toolkit/docs/4.0/security/authzframe/security_descriptor.html.

After a brief introduction, summarizing general guidelines related to the use of Security Descriptors, the appendix consists of some paragraphs that address different features provided by the Java WS Core framework. For each one the different options provided by the framework are evaluated and corresponding guidelines are provided. A final section explains how to describe DILIGENT service security in design documents.

A.1. General guidelines

The Java WS Core allows defining Service security through a Security Descriptor and provides Security APIs to access and modify these information during service execution.

Security configuration of each DILIGENT service is exported in the DILIGENT Information System (DIS) to be used by other Collective Layer services (e.g., VDLGenerator). Modifications of security properties during service lifetime greatly increase complexity and coupling of other Collective Layer services. For these reasons modification of security properties during service lifetime is strongly deprecated.

On the other hand, DILIGENT services are allowed to define multiple Security Descriptors for each service. All Security Descriptors defined for a DILIGENT service are included in the GAR package provided at registration time. At that time, only one of these Security Descriptors is already bound to the Deployment Descriptor of the service, as defining a "default" security configuration for the service. All the Security Descriptors provided with a service are part of the information registered in the DIS-Registry for a given package.

The Keeper is in charge of binding the selected Security Descriptor to the DILIGENT service at deployment time. Only the selected Security Descriptor becomes part of the running Instance profile registered in the DIS-Registry.

Only two properties of Security Descriptors can be modified during service execution: the proxy-file property to add delegated credentials for the service, and the Authorization Chain, to modify authorization handlers.

A.2. Credential Selection & Run As Modes

Credential Selection and *Run As* modes settings for a DILIGENT service (named *Service X* in the following) both depend on the identity scenario adopted by the service. The DVOS service area supports two possible identity scenarios: the *Service Identity* scenario and the *Caller Identity* scenario.

In the *Service Identity* scenario, shown in Figure 59[34], the *Service X* has its own identity and uses it to contact *Other Services*. It acts as a "filter" between *Client*s and *Other Services*, hiding to these services the original caller of a certain request.

This scenario simplifies authorization management because when a new *Client* identity must be entitled to use *Service X* there is no need to entitle it to use *Other Services* also. On the other hand, this scenario reduces the authorization granularity on *Other Services* because an authorized client of *Service X* is automatically entitled to use *Other Services*. In this case, the cardinality of authorization relationships is from N to 1 and from 1 to K[35]. This scenario does not allow entitling a *Client* of *Service X* to use some *Other Services* only.

[34] The notation used does not refer to UML.
[35] N is the number of Clients while K is the number of Other Services.

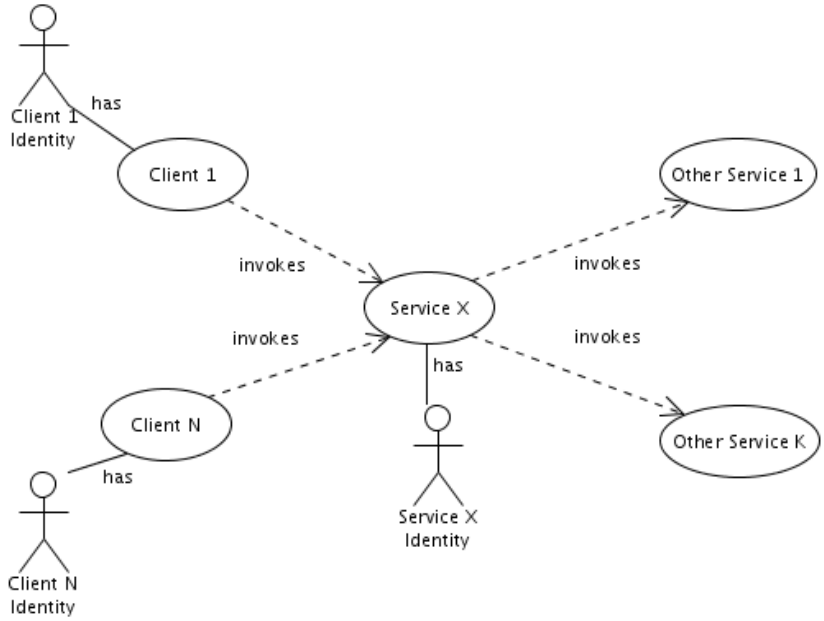

Figure 59. Java WS Core security - Service Identity Scenario

In the *Caller Identity* scenario (shown in Figure 60[36]) the *Service X* does not have its own identity rather it runs with the identity of the callers exploiting the delegation mechanisms. In this case, invocations on *Other Services* performed by *Service X* are done with the identity of *Clients* thus requiring authorization of *Clients* identities on *Other Services*.

In this scenario, the Authorization Management is complicated because of the needs to explicitly entitle a new *Client* identity to use *Service X* and some (or all) *Other Services*. On the other hand the advantage of this scenario is that each *Other Services* can independently choose if a *Client* identity is entitled or not to invoke its operations. In this scenario, the cardinality of authorization relationships is from N to K. This scenario allows entitling a *Client* of *Service X* to use some *Other Services* only.

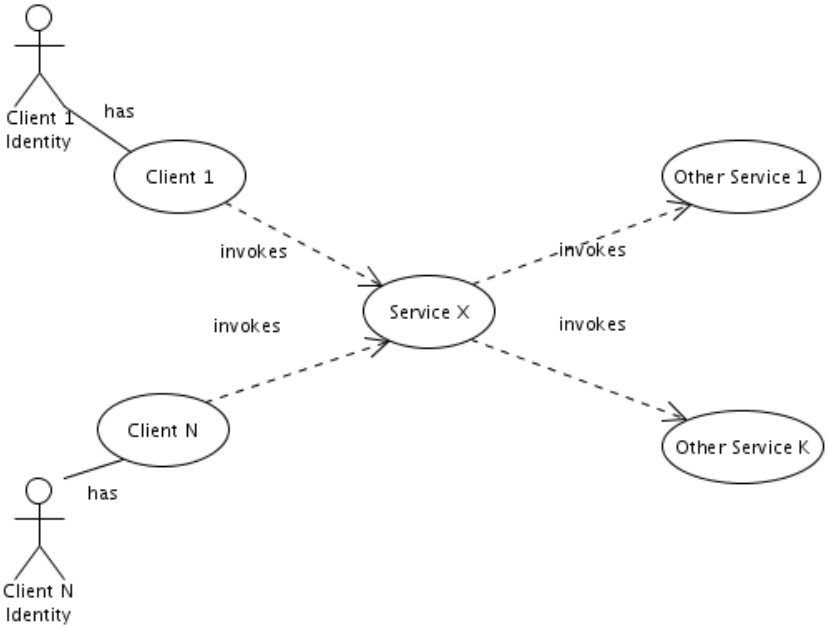

Figure 60. Java WS Core security - Caller Identity Scenario

[36] The notation used does not refer to UML.

The *Service Identity* scenario is most suitable when *Other Services* are highly coupled with *Service X* (e.g. *Other Services* are created and managed by *Service X*) and their behaviors do not depend on the identity of the *Clients*. In a typical use of the *Service Identity* scenario, *Other Services* are not directly visible by *Clients*.

The *Caller Identity* scenario is useful when *Service X* and *Other Services* are loosely coupled (e.g. the lifetime of *Other Services* is independent of the lifetime of *Service X*). In a typical use of *Caller Identity* scenario *Other Services* are directly visible by *Clients,* and they provide functionalities to *Clients* outside the *Service X* mediation.

The explained scenarios refer to a single operation invocation on *Service X*. Different scenarios can be adopted for different operations of *Service X*.

Other more complex scenarios can be adopted (e.g. *Service X* invokes *Other Services* using different identities, or different WS-Resources managed by the same service adopts different identities), but the security features provided by the DVOS do not support these advanced scenarios.

In both the *Service Identity* and *Caller Identity* scenario, the <credentials> and the <proxy-file> tag of the Security Descriptor must be left unset. The name of the file containing proxy credentials to use will be supplied to the service as response of credential creation operation[37]. *Run As Modes* must be set to *Service-Identity* and to *Caller-Identity* respectively. The *System-Identity* option is deprecated; it should be used only by services related to the management of the container itself (e.g.: Hosting Node Manager). The *Resource-Identity* option can be selected, but the service (or WS-resource) itself is in charge of managing the identities associated to WS-resources.

A.3. Authentication Methods

During the choice of authentication methods, service designers should take into account the following considerations.

The *GSITransport* authentication method is deprecated and it should be used only for compatibility reasons. Authentication based on *GSISecureMessage* should be adopted instead.

The use of the *GSISecureConversation* option requires an external Conversation Service (provided as WSRF-Service, but outside the Java WS Core package). A Conversation Service must be available on the node in order to use the *GSISecureConversation* method.

To avoid message tampering the *Integrity* option should be used. The use of the *Privacy* option to protect confidential information should be carefully evaluated. This option introduces a high overhead due to the encryption of messages. The *GSISecureConversation* method should be adopted in order to mitigate the overhead introduced by the message encryption only when the flow of confidential information between two services became substantial.

A.4. Authorization Mechanism

Multiple Authorization handlers can be defined at Service or WS-Resource level. The access to the service (or WS-Resource) is granted if all the handlers defined return a permit.

In the Java WS Core framework the definition of different authorization handlers for different operation of the same service (or WS-Resource) is not allowed. This limitation must be taken into account in designing service or WS-Resource authorization policies.

DILIGENT service designers, basing on the APIs provided by the DVOS Authorization Management, must define VO-level authorization handlers. These handlers will be in charge to enforce VO-level authorization on services (or WS-Resource).

[37] See *Service Credential Consideration*, paragraph 7.1.4.1 of Authentication Support package.

DILIGENT Services or WS-Resources are also allowed to enforce resource specific authorization policies. For this purpose the use of alternatives PDP is also possible. When a PDP is used to enforce service (or WS-Resource) specific authorization its behaviour should be reported in the service (or WS-Resource) documentation, as described in section A.5. Java WS Core also provides some alternatives PDP, if one of these PDPs is used the description is optional.

A.5. Documentation Guidelines

The security configuration of a WSRF Service must be reported in the design document of the service itself. Properties of Security Descriptors defined for a service should not be modified during service execution, hence they must be included in the profile of the service (or WS-Resource).

Following sections must be provided for each Security Descriptor defined for the service (if more than one). Properties of Security Descriptors must be organized as explained below. For each property, allowed options are specified between square brackets.

- **<X>ServiceSecurityDescriptor<N>**: [Default]
  - **Authorization Chain**
    - **Authorization Criteria**
  - For each ***OperationY***
    - **Identity Scenario**: Caller | Service | Other
    - **Authentication Method**: TLS | GSISecureMessage | GSISecureConversation
      - **Protection Level**: Integrity | Privacy
  - **Default Settings**
    - **Identity Scenario**: Caller | Service | Other
    - **Authentication Methods**: TLS | GSISecureMessage | GSISecureConversation
      - **Protection Level**: Integrity | Privacy

Each security descriptor must be contained in a file named **<X>ServiceSecurityDescriptor<N>**; where X is the name of the service and N is a progressive number identifying univocally the security descriptor among those available for that service.

The *Default* option in the **Security Descriptor N** line must be specified in the Security Descriptor bound to the service at registration time only. If only one Security Descriptor is specified for a service, it can be omitted.

The **Authorization Chain** paragraph must be filled with the list of authorization handlers used. If these handlers do not belongs to the set provided by the Java WS Core framework they must be briefly described (including their behaviour, the parameters they require, and any other significant information). The **Authorization Criteria** subsection must contain a description of authorization criteria supposed to be configured for the service. These criteria are related to the use of the service in the DILIGENT environment (e.g.: DILIGENT actors allowed to perform service operations, authorization rules to be enforced, and so on…).

When the *Other* option is selected in the **Identity Scenario**, a subsection including additional details must be provided. This subsection must specify at least credential selection and Run As Mode settings for the operation.

The **Protection Level** must be specified separately for each Authentication Method allowed for the operation.

The **Default Settings** are used for operations not included in the previous sections.

Examples of how to specify these properties can be found in the DVOS section.

In order to enhance service interoperability the D.1.X.2 deliverables must contains at least **Identity Scenarios** adopted for each service operation. Other security options of the previous template can be provided in the D.1.X.3 deliverables.

References

[1] DILIGENT. Deliverable No D1.1.1: "Test-bed Functional Specification"
[2] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. Pattern-Oriented Software Architecture. John Wiley & Sons ISBN 0-471-95869-7, 1996
[3] Ali Arsanjani. Service-oriented modeling and architecture. IBM developerWorks. http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/
[4] Andreaozzi S., Burke S., Field L., Fisher S., Kónya B., Mambelli M., Schopf J., Viljonen M., Wilson A. GLUE Schema Specification (version 1.2). Draft. Last version available at http://infnforge.cnaf.infn.it/docman/?group_id=9
[5] E. Michael Maximilien, Munindar P. Singh A Framework and Ontology for Dynamic Web Services Selection. IEEE Internet Computing, 8(5), 84:93, 2004
[6] S. Farrell, R. Housley. RFC 3281: An Internet Attribute Certificate Profile for Authorization. http://www.faqs.org/rfcs/rfc3280.html.
[7] gLite Team. gLite Installation Guide v1.3 (rev. 3) August 2005
[8] Gamma E., Helm R., Johnson R., and Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley
[9] W3C OWL Web Ontology Language Use Cases and Requirements http://www.w3.org/TR/webont-req/
[10] DARPA Agent Markup Language. OWL-based Web Service Ontology. http://www.daml.org/services/owl-s/
[11] Baader F., Calvanese D., McGuinness D., Nardi D., Patel-Schneider P. The Description Logic Handbook. Cambridge University Press ISBN 0-521-78176-0, 2003
[12] DARPA Agent Markup Language. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.daml.org/rules/proposal/
[13] Dublin Core metadata Initiative web site. http://dublincore.org/
[14] Gonçalves M. A. Streams, Structure, Spaces, Scenarios, and Societies (5S): A Formal Digital Library Framework and Its Applications. Virginia Polytechnic Institute and State University. PhD Dissertation. 2004
[15] Czajkowski K., Ferguson D. F., Foster I., Frey J., Graham S., Sedukhin I., Snelling D., Tuecke S., Vambenepe W. The WS-Resource Framework. White paper, 2004
[16] S. Farrell, R. Housley: RFC 3281: An Internet Attribute Certificate Profile for Authorization, http://www.faqs.org/rfcs/rfc3280.html.
[17] Kerberos: The Network Authentication Protocol, http://web.mit.edu/kerberos/www/
[18] Shibboleth Project, http://shibboleth.internet2.edu/
[19] M. Wahl, S. Kille, T. Howes: RFC 2253: Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names, http://www.faqs.org/rfcs/rfc2253.html
[20] T. Dierks, C. Allen: The TLS Protocol Version 1.0, http://www.ietf.org/rfc/rfc2246.txt
[21] J. Basney, M. Humphrey, V. Welch: The MyProxy online credential repository, http://grid.ncsa.uiuc.edu/myproxy/
[22] OASIS. Web Services Resource Properties 1.2 (WS-ResourceProperties). Public Review Draft 01, 10 June 2005. S. Graham and J. Treadwell Editors. http://docs.oasis-open.org/wsrf-ws_resource_properties-1.2-spec-pr-01.pdf
[23] OASIS. Web Service Topics 1.2 (WS-Topics). Working Draft 01, 22 July 2004. William Vambeneper Editor. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf
[24] Shibboleth Project, http://shibboleth.internet2.edu/
[25] M. Wahl, S. Kille, T. Howes: RFC 2253: Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names, http://www.faqs.org/rfcs/rfc2253.html
[26] T. Dierks, C. Allen: The TLS Protocol Version 1.0, http://www.ietf.org/rfc/rfc2246.txt

[27] J. Basney, M. Humphrey, V. Welch: The MyProxy online credential repository, http://grid.ncsa.uiuc.edu/myproxy/
[28] P. Leach, M. Mealling, R. Salz: A Universally Unique IDentifier (UUID) URN Namespace, http://www.ietf.org/rfc/rfc4122.txt
[29] S. Graham, D. Hull, B. Murray: Web Services Base Notification 1.3, http://www.oasis-open.org/committees/download.php/13488/wsn-ws-base_notification-1.3-spec-pr-01.pdf
[30] Vincenzo Ciaschini: VOMS Core Services, https://edms.cern.ch/file/571991/1/voms-guide.pdf
[31] D. W. Chadwick, O. Otenko, The PERMIS X.509 Role Based Privilege Management Infrastructure, http://sec.isi.salford.ac.uk/download/SACMATfinal.pdf