Information Society
Technologies

Diligent

A DIgital Library
Infrastructure on Grid
ENabled Technology

Deliverable No D1.2.1:

"DL Creation & Management Services Specification interim report"

May 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:	D1.2.1
Deliverable title:	DL Creation & Management Services Specification interim report
Contractual Date of Delivery:	Month 8
Actual Date of Delivery:	15 June 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:	14
Distribution:	Confidential
Nature:	Report
Version/Revision:	1.0
Draft/Final	Final
Total number of pages: (including cover)	104
File name:	D1.2.1-FinalVersion.doc (Submitted to EC as D1.2.1.doc)
Key words:	*Digital libraries, UML, Services Architecture, Services Design*

Disclaimer

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 within the first period of 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 - May 31st 2005.

By analyzing the functions and features of the DILIGENT system, as reported in *D1.1.1 Test-bed functional specification*, the *DL Creation & Management Services Specification interim report* i) identifies those related with the services forming this functional area, ii) specifies the functionalities offered by each service, and iii) presents the architectural overview of them 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 *D1.2.1 DL Creation & Management Services Specification interim report* is to present the first 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 further organized into three sub-phases whose final goal is to produce a detailed service specification that will guide the subsequent services implementation, testing, and maintenance phases. This goal will be achieved by producing three reports, each adding further level of detail to the specification. These reports are respectively i) this interim report, ii) the *D1.2.2 DL Creation & Management services specification report* due in July 2005, and finally iii) the *D1.2.3 DL Creation & Management services detailed design report* due in January 2006.

In particular this report presents the architectural overview of each of the cited services in order to capture and convey on the most significant architectural decisions that have been made about the services. These decisions will also influence the design of the other services of the DILIGENT infrastructure.

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 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 service. Then the constraints imposed by the environment in which the system/software must operate, by the need to reuse existing assets, and by the imposition of various standards 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 we want the system to support has 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:

a) Boundary class, entity class, control class or any other class that is deemed as relevant to better understand the design of the service and the interaction with other DILIGENT services and/or gLite services;
b) Component, library component, subsystem component or any other component that is deemed as relevant to better understand the design of the service and the interaction with other DILIGENT services and/or gLite services.

Finally, taking into account that this report has been conceived as the first 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 D1.2.2/3]*. The information reported in the section are the result of the initial specification phase and more detail information will be added in the:

   i. Final service specification (D1.2.2), if this information is related to a more understandable description, to the specification of new control or entity classes, to the specification of new subsystem component;

   ii. Detailed design (D1.2.3), if 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 D1.2.2/3*], if an in depth investigation is still needed. 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 class of services) are 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 first period of the various 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) this interim report, ii) the *D1.2.2 DL Creation & Management services specification report* due in July 2005, and finally iii) the *D1.2.3 DL Creation & Management services detailed design report* due in January 2006.

The services belonging to the WP1.2, objective of this report, are those responsible for providing functionalities related with the creation and maintenance of virtual digital libraries.

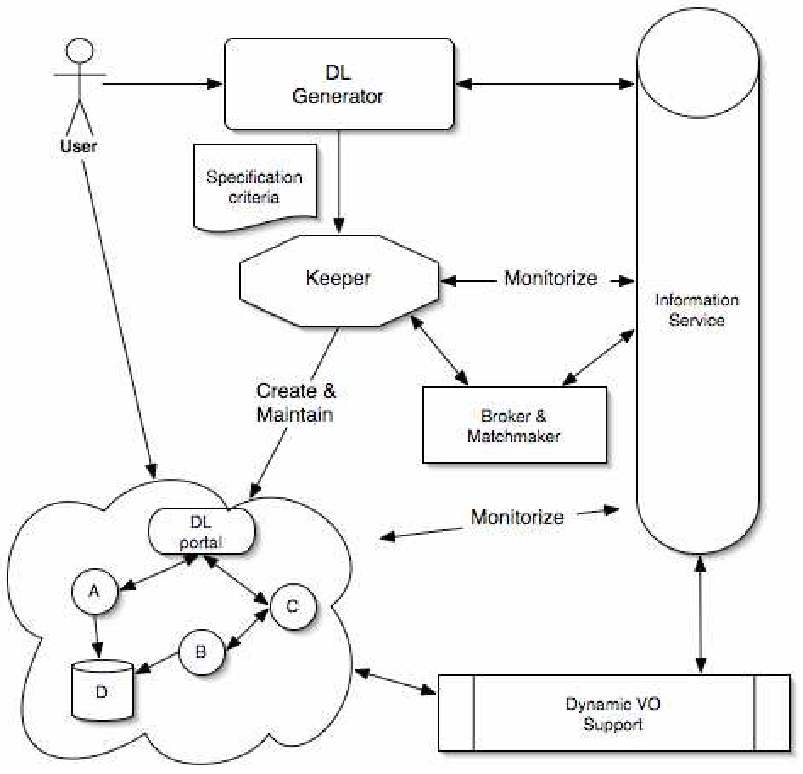


Figure 1. DL Creation and Management services

In Figure 1 are shown these services and the main interactions among them:

- The *Information Service* gathers and supplies information about any DILIGENT resource;
- The *Broker & Matchmaker Service* identifies the most suitable hosting nodes among those available where new DILIGENT service instances can be deployed;
- The *Keeper Service*, which is responsible to physically create the virtual digital library and maintain it operational by appropriately sharing available resources and creating new service instances;
- The *Dynamic VO Support Service*, which enables the creation of virtual organization in order to support the controlled and trusted sharing of resources within the system;
- The *VDL Generator Service*, which supplies to users the mechanism to define the characterization criteria about the DL they are interested in, and thus initiates the DL creation process.

Therefore, WP1.2 is responsible for the entire design and realization of the DILIGENT Collective Layer (plus the VDL Generator Service strictly related to this layer). This report includes the first specification of the DILIGENT Collective Layer services. It aims to provide to WP1.3 – WP1.6 the basic and necessary knowledge needed to conduct their design phases.

The outline of this report is as follows: the next section presents the rationale of the services specification interim 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; finally, Section 9 concludes reporting the follow up of this report, in particular its relationships with *D1.2.2 DL Creation & Management services specification report*.

2 RATIONALE OF SERVICES SPECIFICATION INTERIM 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 will be 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 deliverable produced by the WP1.2, *D1.2.1 DL Creation & Management Services Specification interim report,* is particularly important document because it represents the first design document of the services forming the DILIGENT Collective Layer. These services are of global utility in the sense that they are responsible for supplying the basic functionalities to manage the DILIGENT system or, in other words, they act as the glue of the whole federation of resources forming the DILIGENT infrastructure. As a consequence, this report has been made available to the other services' designers because the architectural choices reported in it influence the design of all the DILIGENT services.

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, 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 nature constraints there 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, using a number of different architectural views to depict different aspects of it. It is 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, at least the following view sections are provided:

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 only 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 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.

Detailed information about any significant class has been presented by adding a subsection whose content is described in Section 2.2.4. For instance, it has been used to describe the algorithm represented by the class, the kind of data manipulated by that class and any other kind of detailed information deemed as relevant. In particular, it **has** been described in detail:

- Each boundary class in order to improve the definition of the boundary of each service. This allows to have a detailed description of the user interface and access method that the class offers to the surround;
- Each entity class or control class that is deemed as relevant also for other DILIGENT services.

With respect to use-case realization the goal has been to illustrate how the system works explaining how the various designed elements collaborate to the realization of 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 may be 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 to provide 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 Significant Classes and Components

2.2.4.1 Class XXX

A section for each boundary class, entity class, control class or any other class that is deemed as relevant to better understand the interaction with other DILIGENT services and gLite services, assumptions, constraints and any kind of detail that is deemed as relevant.

2.2.4.2 Component YYY

A section for each service component, library component, subsystem component or any other component that is deemed as relevant to better understand the interaction with other DILIGENT services and gLite services, assumptions, constraints and any kind of detail that is deemed as relevant.

3 DILIGENT RESOURCES MODEL

A 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 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 resource owner registers a resource, he/she will also provide a description of it. The services that implement the Resource Management functionalities presented in *D1.1.1 Test-bed functional specification* report will complete this description by automatically gathering additional information, if possible. The full description will be stored in an XML format and will be maintained both as information handled by the services of the Collective Layer and as a special materialized collection managed by the services of Content Management. Storing these resource descriptions in a special collection will permit to extend the basic discovery capabilities provided by the Resource Management (mainly the browse and search functionality based on exact matching procedures) with the advanced search functionalities that will be supported through the standard Search capabilities. This latter kind of storage requires that a DL capable to support the DILIGENT project will be up and running.

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 *Resource* is characterized by a unique identifier, allowing to unambiguously identify it, and a type, allowing to discriminate by the 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 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 monitoring the resource to measure the parameter;
- 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 D1.2.2.

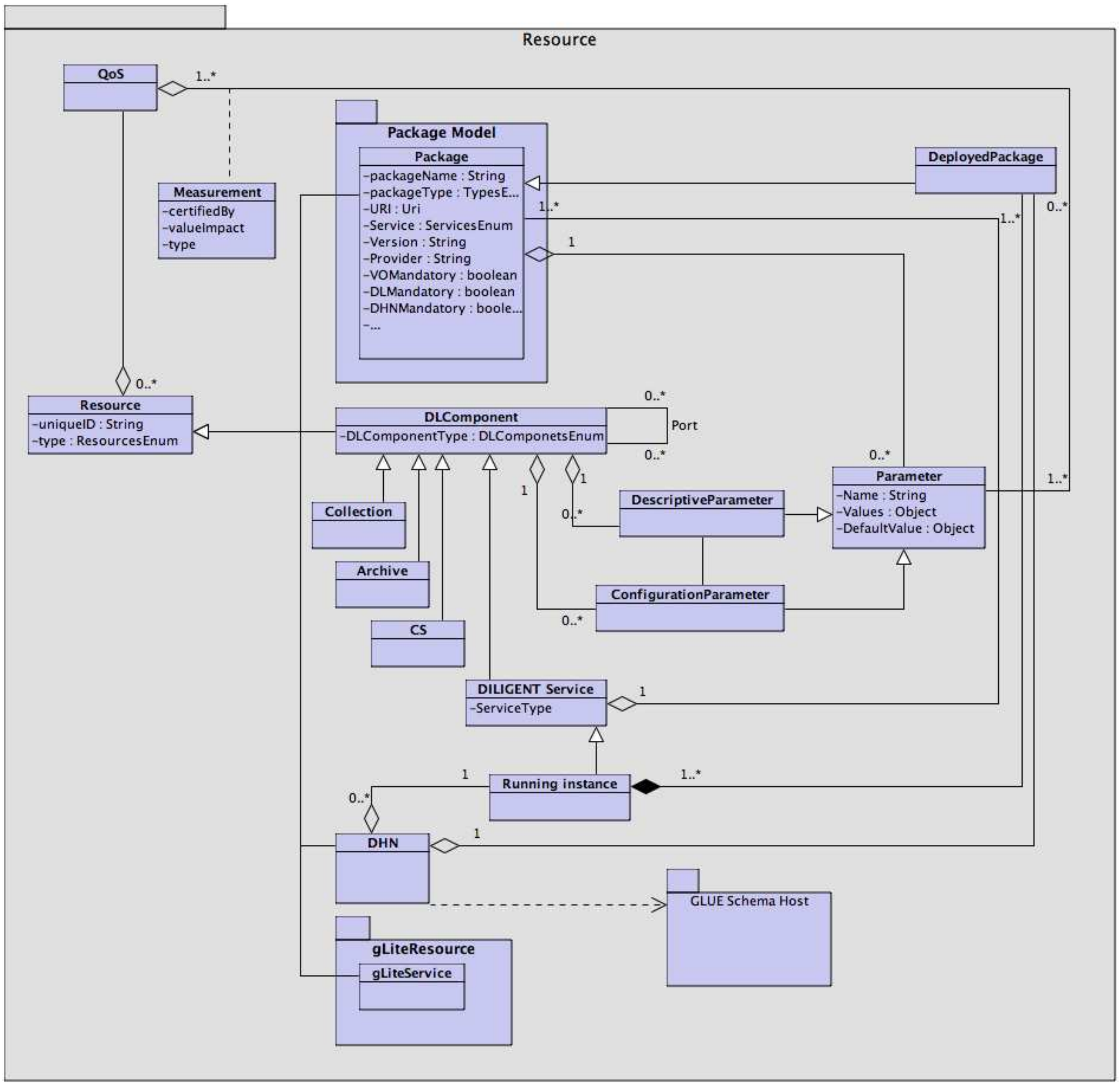


Figure 2. The DILIGENT Resources Model

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 will mainly use all the information presented in figure plus other presented by the Package Model in Section 3.1. More or less, the information maintained about packages are the expected ones i.e. its name, the type, the URI where the package can be found, the

service it belongs to, the version, the provider, etc. Moreover, for each package the system will maintain 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 building a DILIGENT Service.

2. *DLComponent*. This class of resources represents the components that can be used to build a DL. Contrary to packages, these resources represent the logical resources that grouped together form a DL. In fact, a DL Component, as a DILIGENT Service is, aggregates a set of packages.

   As described in detail in Section 8, 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 D1.2.2;
   - 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. relationship with other DLComponents that enable 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 D1.2.2. Figure 2 also highlights a particular type of DLComponents: *Running Instance*, i.e. a particular DILIGENT Service because it represents a significant subpart of DL Components and need to be treated in a particular way (e.g. it could be shared among VOs).

3. *DHN*. It represents a hosting node, i.e. a computer machine connected to the network, which is available within the DILIGENT infrastructure and is capable to host DILIGENT Services. For the time being, we figure out that the information maintained about this kind of resources 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 software that will be registered to the DILIGENT Infrastructure. A data model for these resources is reported in Section 3.2.

3.1 Package Model

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

- *Descriptive information*, like name, type, URI, version, provider.
- *Deployment information*. This second type of information can be further divided into two subtypes:

[2] This name derives from the *component-port* approach adopted in configuration systems. More details are given in Section 8.

    - *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 driving the Keeper on the choice of the DHN where the package can be deployed.

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

Within the DILIGENT context four types of packages have been identified:

- *WSRFService*, representing a package that once deployed produces a Running Instance of a DILIGENT Service. For each WSRFService, the system captures information about the single files constituting it, install/uninstall scripts, etc.
- *Portlet*, representing a package that once deployed produces a portlet that can be hosted by the DILIGENT Portal;
- *Executable*, representing a package containing the code and the related information needed to run a certain job;
- *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 to be introduced on the usage of the model within the DILIGENT system see Section 6.4.2.

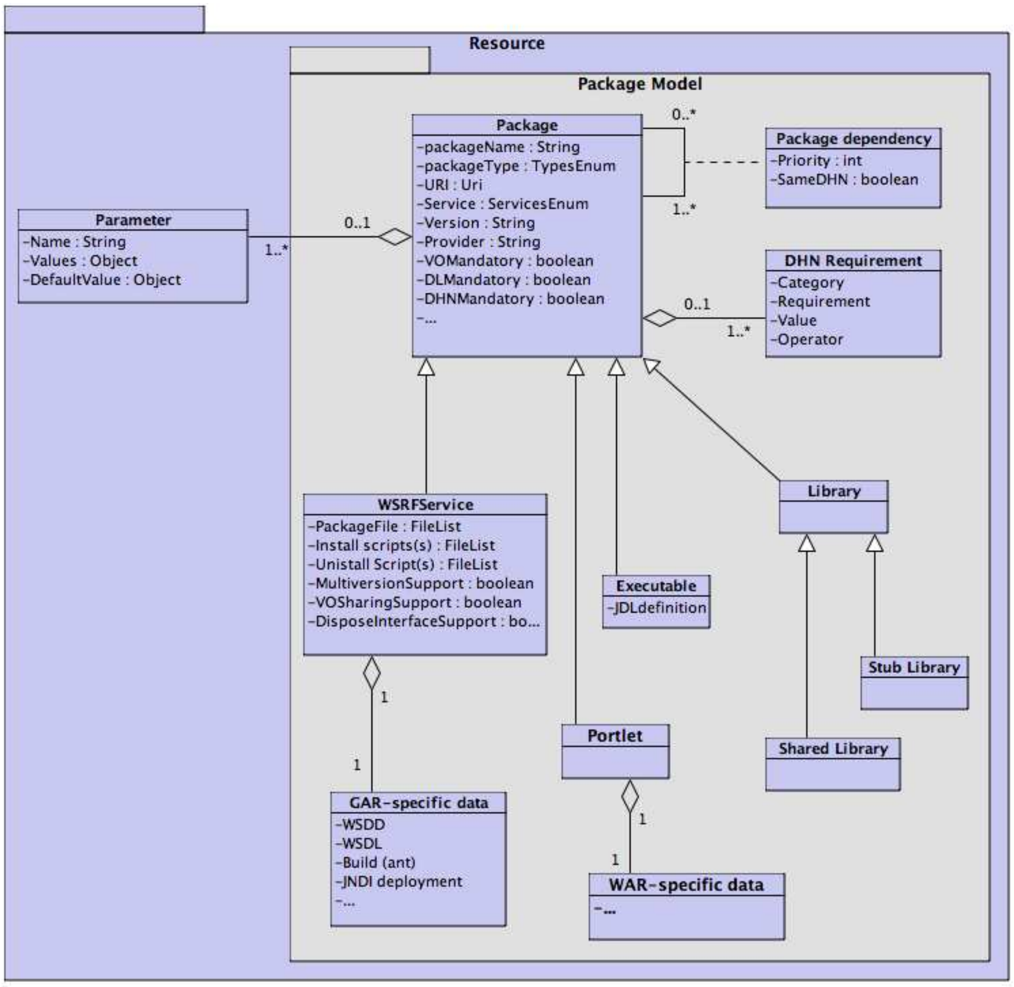


Figure 3. Package Model

3.2 gLite Resource Model

In Figure 4 [To be detailed in D1.2.2] we report the structure and the main characteristics on the information maintained by the system about the gLite resources. For the time being and due to the ongoing design phase of the gLite middleware we decided to report here 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).

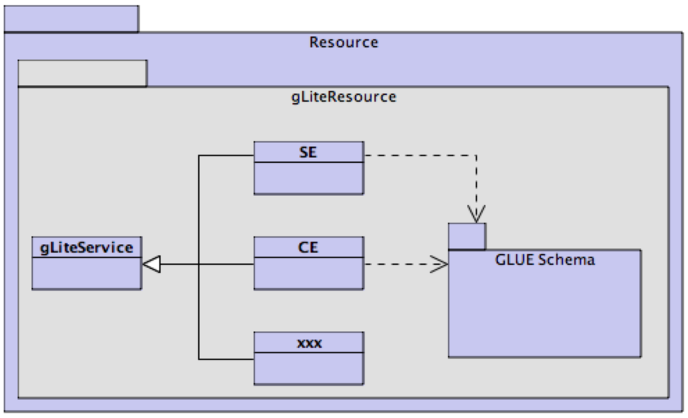


Figure 4. gLite Resource Model

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, we plan to investigate the need to enrich that pre-existing information with other attributes as well as appropriate mechanisms to do it; for instance, we plan to investigate the appropriate procedure for QoS parameters estimation of gLite resources.

[To be provided in D1.2.2]

3.3 DILIGENT Security Model

[To be provided in D1.2.2]

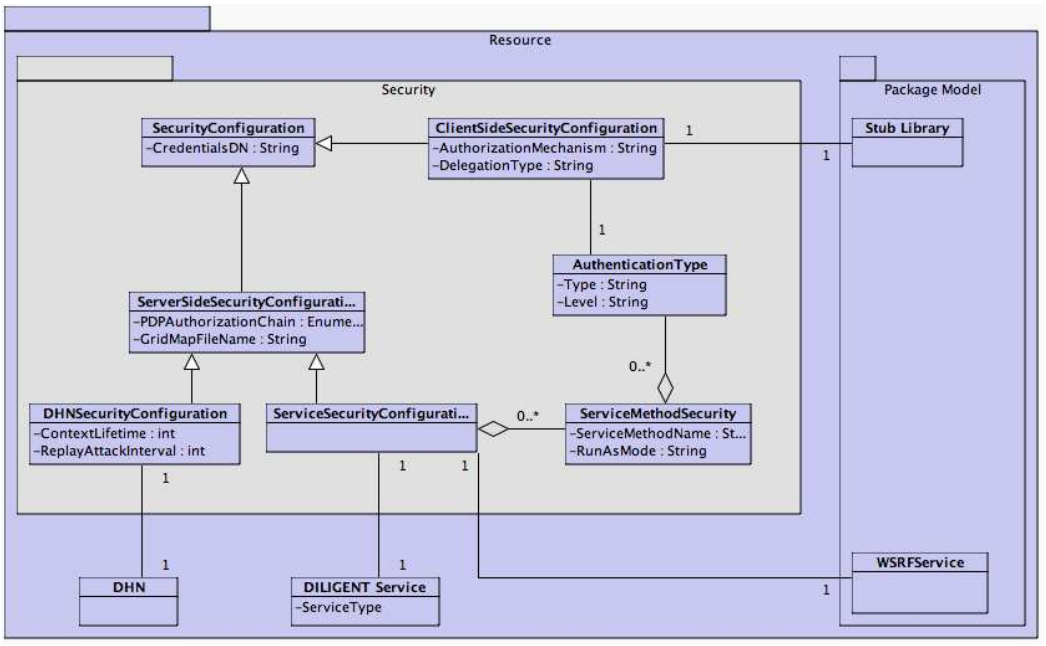


Figure 5. DILIGENT Security Model

4 INFORMATION SERVICE

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 distributed resources that compose the DILIGENT VO.

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.

The DIS model is based on the Grid Monitoring Architecture (GMA) approach proposed by GGF[3] and the Monitoring and Discovery System (MDS) implemented in the Globus Toolkit[4]:

- 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).
- MDS is a set of web services that monitor and discover resources on Grids, allowing users to discover resources in a Virtual Organization (VO) and to monitor those resources.

The rest of the section is organized as follows: in the next section we present the DIS use-case view showing how producers and consumers interact with the DIS, and how the DIS is organized to gather resources information, to monitor resources, and to disseminate that information to the DILIGENT services; In Section 4.2 we present a logical view introducing the DIS architecture in terms of packages and classes representing the main components; In Section 4.3 a deployment view is used to describe how DIS components are deployed on the DHNs; finally, in Section 4.4 a detailed description of the most important classes and components is reported.

Terminology

In the following:

1. a *resource profile* is a set of information provided at registration time compliant to the DILIGENT Resource Model;
2. a *resource status information* is any data related to the current status of a resource conformed to a specified format;
3. a *generic resource information* is intended as any combination of 1. or 2.;
4. a resource status information is *published* in the DIS when the resource adopts the push model;
5. a resource status information is *provided* to the DIS when the resource adopts the pull model;

4.1 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

[3] http://www.ggf.org/ and http://www-didc.lbl.gov/GGF-PERF/GMA-WG/

[4] http://www.globus.org/toolkit/

- 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 providers (DIS-RP - stands for DIS-Resource Provider), of the resource consumers (DIS-HLS – stands for DIS-High-Level Service), and for resource information collection (DIS-IC – stands for DIS-Information Collector):

- Producers are DIS components that publish/provide information. A producer is always linked with a DILIGENT resource that uses it to publish its status information in the DIS. The communication between the resource and its producer can adopt both the push and the pull model. They are classified into two categories: WSRF compliant producers (e.g. used by the DILIGENT Index service), that publish information following the WS-ResourceProperties standard[5], and non-WSRF compliant producers (used by external legacy components), that publish their information in a proprietary format. In the push model, a producer publishes information by performing the appropriate use-case ("Publish WSRF compliant resource information" and "Publish ad-hoc resource information"). In the pull model, resources are called by the DIS ("Update resource information" use-case) to provide their state information. The DIS provides the functionality to wrap non-WSRF compliant resource information, like the information from gLite[6], through the use-case "Format resource information".

- Consumers are DILIGENT services that consume the information. To support consumers the DIS offers the high-level services (DIS-HLS) package of functionalities that provides the following functionalities: discovery, notification, get status and visualization. These functionalities have a customized level of authorization policies by exploiting the "Enforce Resource Policy" use-case of the DVOS service (see Section 7).
- Between the producers and the consumers there is the package DIS-IC that plays the role of the Registry in the GMA model and provides the functionality to collect and register resource information.

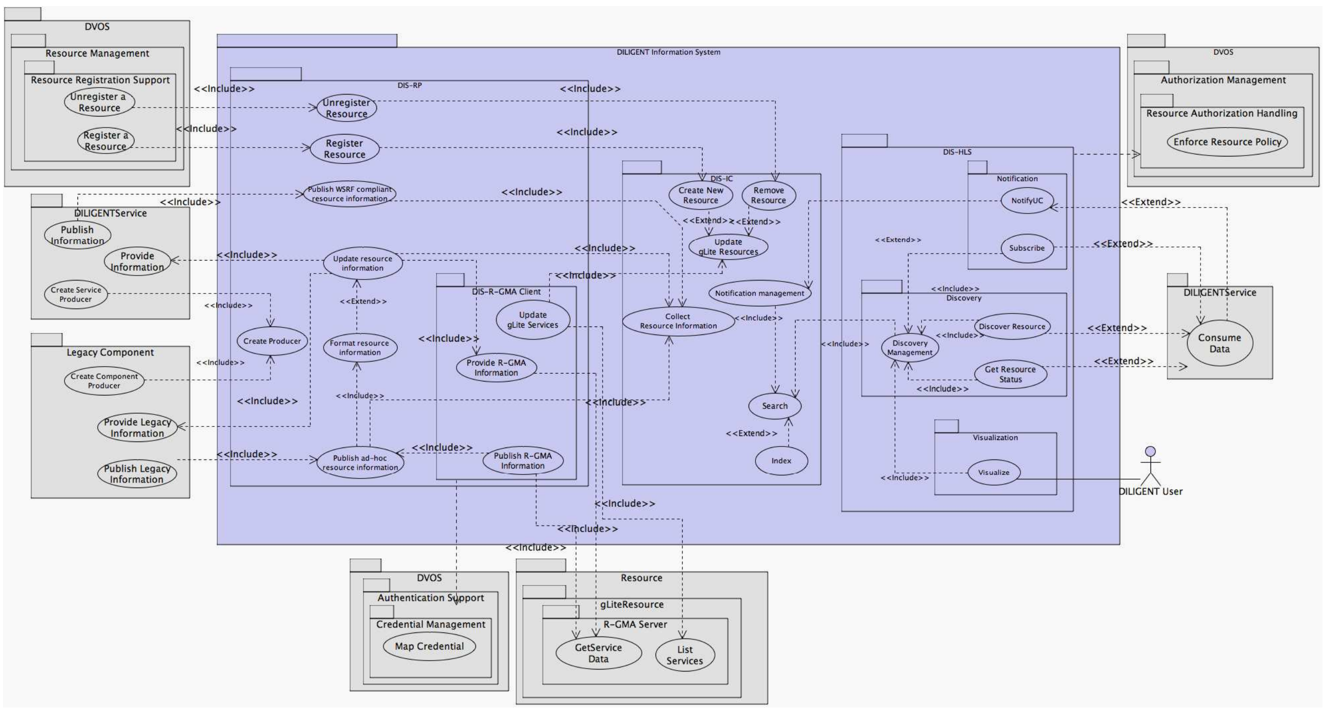


Figure 6. DIS Use-Case view

[to be detailed in D1.2.2]

Resource Providers package (DIS-RP)

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

When a new resource is registered by the DVOS into the DIS, the "Register a Resource" use case uploads the resource profile. At that time, it must be specified the information-updating model adopted by the resource. If the push model is selected then the resource performs according to the "Publish WSRF compliant resource information" or "Publish ad-hoc resource information" use cases to update its resource information. If the pull model is selected then the "Update resource information" use case is performed by the DIS with a configurable interval in order to call "Provide Information" or "Provide Legacy Information" use cases (both use cases are respectively provided by DILIGENT Services or by Legacy Components).

- **Register Resource and Unregister Resource** – The DVOS performs these use-cases to add and delete a resource from the DIS.
- **Create Producer** – A service performs this use-case to create a new producer to use to publish its state into the DIS.
- **Publish WSRF compliant resource information** – By performing this use-case a DILIGENT Service publishes its state into the DIS.
- **Publish ad-hoc resource information** – By performing this use-case legacy components publish their state into the DIS.
- **The DIS-R-GMA Client** sub-package is responsible for collecting information about gLite services. "Update gLite Services" use-case retrieves from R-GMA the list of available services and their status information. Then, via the "Publish RGMA Information" and "Provide RGMA Information" use cases this information is published/provided in the push and the pull model respectively.
- **Update resource information** – This use-case is performed periodically by calling all resources that use the pull publication model, i.e. include the "Provide Information" use-case, if they are DILIGENT Services, or the "Provide Legacy Information" use-case, if they are legacy components. In the second case, if it is necessary to wrap non-WSRF compliant resource information, the "Format resource information" use-case is called.
- **Format resource information** – This use-case is performed to wrap non-WSRF compliant resource information into WS-ResourceProperties.

Information Collector package (DIS-IC)

- **Create new Resource, Remove Resource, and Update gLite Resources** – These use-cases add and delete a resource profile (conforming to the DILIGENT Resource Model) from/to the DIS; in particular "Update gLite Resources" updates the list of available resources from the information gathered from R-GMA.
- **Collect Resource information** – It is a use-case that is in charge to store and provide a uniform access to all DILIGENT resources status information.
- **Search and Index** – Indexes over DIS resources information status are built by the "Index" use case (representing also the access to that index) in order to support queries from the "Search" use case.
- **Notification Management** – It models the notification mechanism offered by the DIS.

High-level services package (DIS-HLS)

Note that executions of all use-cases of this package are subject to authorization from the DVOS.

- **NotifyUC and Subscribe** –A DILIGENT service can register itself to events notification by performing the "Subscribe" use-case, i.e. it subscribes to be notified when the information status of a defined resource changes. The "NotifyUC" use-case models the notification events delivered to DILIGENT services according to the subscriptions performed by them.
- **Discover Resource, Get Resource Status and Discovery Management** – These use-cases are grouped into the Discovery package to allow DILIGENT services to discover resources and gather their status information. To discover resources the DILIGENT services perform queries by specifying resource characteristics, i.e. any information available from the DIS-IC package related to a resource profile or status.
- **Visualize** – The Visualization package contains a use-case that allows an authorized DILIGENT user to browse and view the resources and their status information.

4.2 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. Following a brief description of each package is provided (see Section 1.4 for a detailed specification of each package).

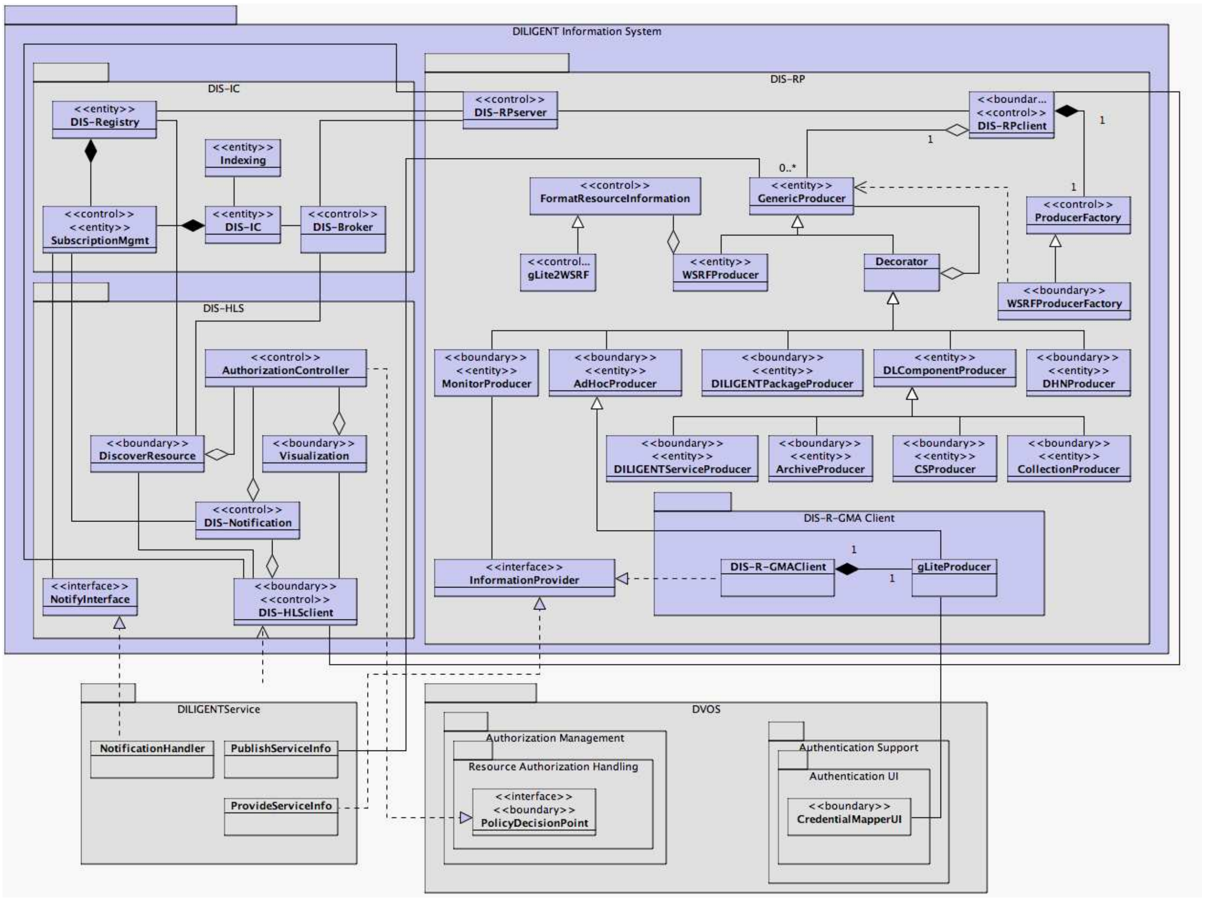

DILIGENT Information System
DIS-IC
<<entity>> DIS-Registry
<<entity>> Indexing
<<control>> <<entity>> SubscriptionMgmt
<<entity>> DIS-IC
<<control>> DIS-Broker
DIS-HLS
<<control>> AuthorizationController
<<boundary>> DiscoverResource
<<boundary>> Visualization
<<control>> DIS-Notification
<<interface>> NotifyInterface
<<boundary>> <<control>> DIS-HLSclient
DIS-RP
<<control>> DIS-RPserver
<<boundar... <<control>> DIS-RPclient
<<control>> FormatResourceInformation
<<entity>> GenericProducer
<<control>> ProducerFactory
<<control... gLite2WSRF
<<entity>> WSRFProducer
Decorator
<<boundary>> WSRFProducerFactory
<<boundary>> <<entity>> MonitorProducer
<<boundary>> <<entity>> AdHocProducer
<<boundary>> <<entity>> DILIGENTPackageProducer
<<entity>> DLComponentProducer
<<boundary>> <<entity>> DHNProducer
<<boundary>> <<entity>> DILIGENTServiceProducer
<<boundary>> <<entity>> ArchiveProducer
<<boundary>> <<entity>> CSProducer
<<boundary>> <<entity>> CollectionProducer
DIS-R-GMA Client
DIS-R-GMAClient
gLiteProducer
<<interface>> InformationProvider
1
0..*
DILIGENTService
NotificationHandler
PublishServiceInfo
ProvideServiceInfo
DVOS
Authorization Management
Resource Authorization Handling
<<interface>> <<boundary>> PolicyDecisionPoint
Authentication Support
Authentication UI
<<boundary>> CredentialMapperUI

Figure 7. DIS Logical View

DIS Information Collector package (DIS-IC)

The classes of this package provide uniform access to DIS resource information.

- **DIS-Registry** – It stores the profiles of all registered resources. It also processes queries about these profiles from the DiscoverResource class.
- **DIS-Broker** – It decides which, among the available information collectors (DIS-IC class) has to be used to publish specific resource status information.
- **DIS-IC** – It stores all the DILIGENT resource status information, by accepting them from producers (via the DIS-RPClients) in a WS-ResourceProperties compliant format. It also manages the queries from DIS high-level classes.
- **Indexing** – It models indexes about all the collected resources status information.
- **SubscriptionMgmt** – It models the dispatch mechanism for delivering notification to subscriber.

[to be detailed in D1.2.2]

DIS Resource Provider package (DIS-RP)

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

- **ProducerFactory, WSRFProducerFactory** – These classes are responsible for the creation of new producers and for the notification to the DIS-Registry about the created producers; they are modeled following the Factory pattern [8] that provides an interface for creating families of related producer objects without specifying their concrete classes. A new DIS producer is created when a new resource is deployed. At creation time the publication model to be used, push or pull, must be declared.
- **InformationProvider interface, GenericProducer, and all the classes that inherit from it** – A DIS producer is an instantiation of some classes derived from the GenericProducer root class, according to the producer type. Those classes are in charge to accept resource status information from the related DILIGENT resource and send it to the collector (DIS-IC class). Producer classes follow the Decorator pattern [8] that attaches additional responsibilities to an object dynamically, e.g. when the publication model (push and pull) is changed after resource creation the producer objects need to respond to that configuration change.

  If the push model has been selected, a DILIGENT service accesses to its local producer each time it wants to change its published status information.

  The MonitorProducer class is a particular producer class that is responsible for periodically sending requests to a resource that has selected the pull model in order to update resource status information maintained by the DIS-IC. DILIGENT services that want to use the pull model must implement the InformationProvider interface that defines the protocol that is used between MonitorProducer class and the DILIGENT service itself.
- **DIS-RPClient** – DIS-RPClient class is responsible for collecting status information from resource producers located in the same DHN and forwarding them to the DIS-RPServer. It contacts the local DIS-HLSClient in order to discover which DIS-RPServer to access.
- **DIS-RPServer** – The DIS-RPServer is in charge of pushing resources information to the DIS-Registry (in the case of a registration or deletion of a resource) or to the DIS-Broker (in case of a resource status information). It contacts the local DIS-HLSClient in order to discover which DIS-Registry and DIS-Broker to access.
- **FormatResourceInformation** – This class models a generic wrapper for a non-WSRF resource allowing to obtain status information in WS-ResourceProperties

format. It implements the Strategy pattern [8] that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows to wrap different non-WSRF compliant resources by adopting the same class.

- **gLite2WSRF** – This class models a wrapper that transforms gLite resource descriptions into a WS-ResourceProperties format storable in the DIS-IC

[to be detailed in D1.2.2]

DIS high-level services package (DIS-HLS)

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

- **DiscoverResource** class is responsible for providing a way to query the resource profiles registered in the DIS-Registry.
- **DIS-Notification** class models the subscription/notification mechanism that the DILIGENT services can use to be notified about events concerning status and profiles. This class implements the Observer pattern [8] that defines a one-to-many dependency between objects so that when one resource changes state, all its dependents are automatically notified.
- **NotifyInterface** is the interface that any DILIGENT service must implement in order to be notified by the DIS.
- **Visualization** class is responsible to provide to DILIGENT users a view of the resource information maintained in the DIS-IC package.
- **AuthorizationController** class implements the PolicyDecisionPoint interface required by the DVOS service in order to have a customized level of authorization policy in the DIS.
- **DIS-HLSClient** is the access point to all DIS-HLS classes.

[to be detailed in D1.2.2]

4.3 Deployment View

The DILIGENT Information Service is decomposed in the following components:

- **DIS-RP Client** – It is a library that must be present on each DHN (so its package is labelled as "DHNMandatory" at registration time in the Packages Repository). This library includes the producer factory classes and all the producers classes depicted in Figure 7. It must be used by each DILIGENT service in order to create the appropriate producer to use to publish/provide information status into/to the DIS-IC. For instance, the Hosting Node Manager of the Keeper that resides in a DHN must request to the DIS-RP Client the creation of a local DHNProducer to use to publish the DHN status information. Again, a service that wants to use the pull model must request to the DIS-RP Client the creation of a MonitorProducer and implement the InformationProvider interface to provide status information.
- **DIS-RP Server** is a WSRF service (registered in the Packages Repository – Section 6.4.2 - as WSRFService package) that collects information from producers and sends it to the DIS-Registry (if they are related to resource profiles) or to DIS-IC using the DIS-Broker (if they are related to the resource status).

- **DIS-R-GMA Client** is a WSRF service (registered in the Packages Repository as WSRFService package) that is in charge of harvesting profiles and status from a R-GMA Server about resources of a gLite-based infrastructure (CEs, SEs, I/O Servers, Catalogs, etc.). To publish this information it instantiates (via the local DIS-RP Client) a gLiteProducer.
- **DIS-Registry** is a WSRF service (registered in the Packages Repository as WSRFService package) that contains the profiles of all registered resources. It includes a notification subsystem that receives subscriptions related to resource profiles and sends notifications when they change.
- **DIS-IC** is a WSRF service (registered in the Packages Repository as WSRFService package) and it is in charge of handling the resource status information. It includes a notification subsystem that receives subscriptions related to resource status information and sends notifications when they change. This is a distributed service that means that different ICs manage different resource status information. The logic of this distribution (e.g. the IC/ICs to publish in, the number of replicated ICs to maintain) is implemented in the DIS-Broker.
- **DIS-Broker** is a WSRF service (registered in the Packages Repository as WSRFService package). It is a sort of proxy that distributes resources status information among the available information collectors and mediates the related queries. Of course, it also distributes the subscription requests.
- **DIS-HLS Client** is a library that must be present on each DHN (so its package is labelled as "DHNMandatory" at registration time in the Packages Repository). It is used by local services to access to the DIS-DiscoverResource service in order to discover resources in the DILIGENT VO and to access to their status.
- **DIS-DiscoverResource** is a WSRF service (registered in the Packages Repository as WSRFService package) that is used by the DIS-HLS Client in order to interact both with the Registry and the Broker to consume resource information.
- **DIS-Visualization** component is a portlet (registered in the Packages Repository as Portlet package) that allows an authorized DILIGENT user to navigate the resource information maintained both in the DIS-Registry and in the DIS-ICs.

In Figure 8 we present a possible deployment scenario of the components involved in the publishing information phase.

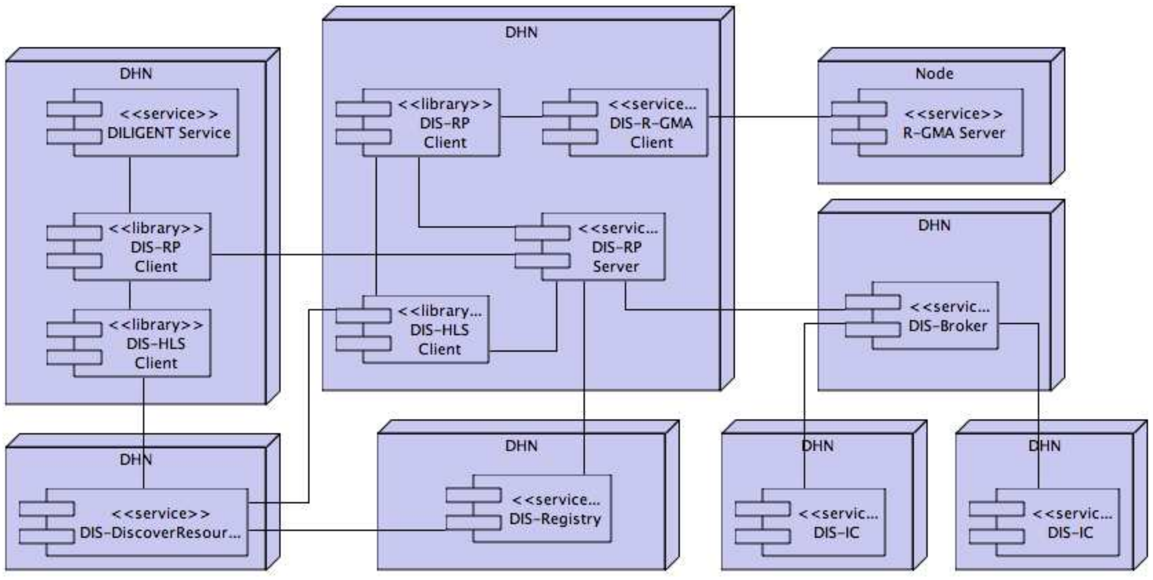

DHN
<<service>> DILIGENT Service
<<library>> DIS-RP Client
<<library>> DIS-HLS Client
DHN
<<library>> DIS-RP Client
<<service... DIS-R-GMA Client
<<servic... DIS-RP Server
<<library... DIS-HLS Client
Node
<<service>> R-GMA Server
DHN
<<servic... DIS-Broker
DHN
<<service>> DIS-DiscoverResour...
DHN
<<service... DIS-Registry
DHN
<<servic... DIS-IC
DHN
<<servic... DIS-IC

Figure 8. DIS Deployment View: Publishing DILIGENT and gLite services status information

A DILIGENT Service contacts the local DIS-RP Client to create its own producer. Then, in order to publish the service status information, the DIS-RP Client contacts the DIS-RP Server and sends it the data. If it does not know where the DIS-RP Server is or it is unreachable, it queries the local DIS-HLS Client in order to discover an available DIS-RP Server. The DIS-HLS Client obtains these information by accessing to the DIS-DiscoverResource service.

Once the resource status information is on the DIS-RP Server, this service contacts the local DIS-HLS Client in order to have access to the DIS-DiscoverResource; By using the DIS-DiscoverResource it discovers the appropriate DIS-Broker to access. Then the resource status information is sent to the DIS-Broker that selects the DIS-IC where the information is maintained.

Figure 8 also shows how the DIS-R-GMAClient service interacts with the R-GMA Server to gather gLite services information and to publish this information into the DIS-Registry and DIS-IC through the local DIS-RP Client.

[to be detailed in D1.2.2]

In Figure 9 we present a possible deployment scenario of the components involved in the consuming information phase.

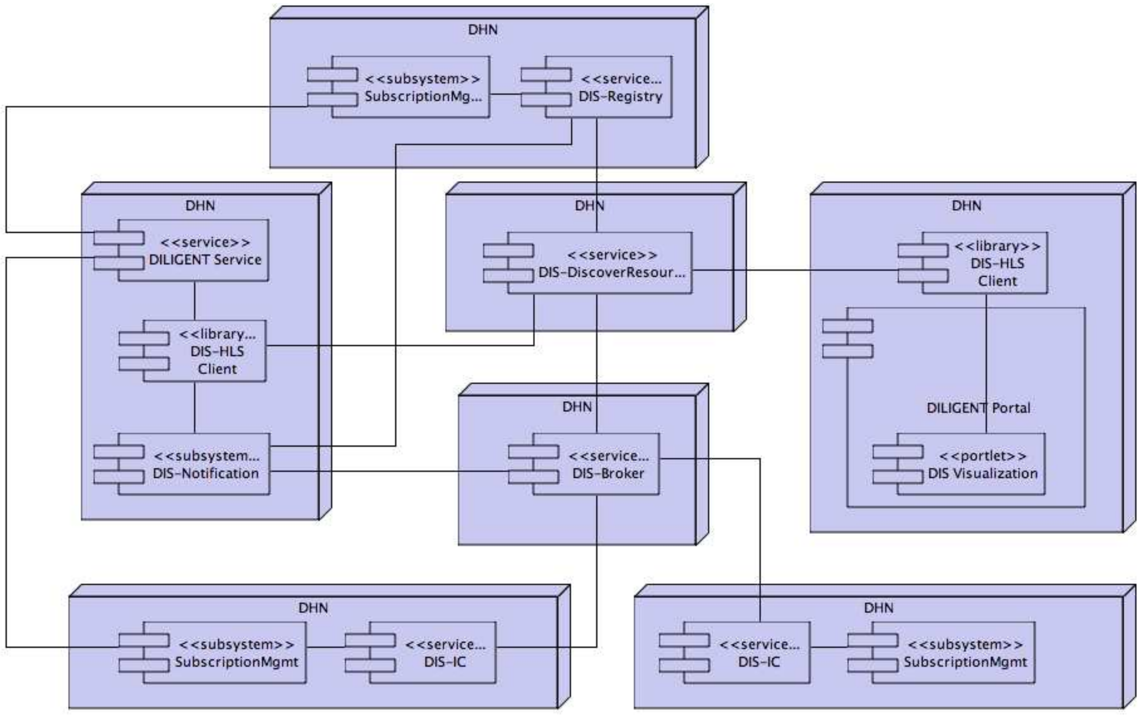

DHN
<<subsystem>> SubscriptionMg...
<<service... DIS-Registry
DHN
<<service>> DILIGENT Service
<<library... DIS-HLS Client
<<subsystem... DIS-Notification
DHN
<<service>> DIS-DiscoverResour...
DHN
<<library>> DIS-HLS Client
DILIGENT Portal
<<portlet>> DIS Visualization
DHN
<<service... DIS-Broker
DHN
<<subsystem>> SubscriptionMgmt
<<service... DIS-IC
DHN
<<service... DIS-IC
<<subsystem>> SubscriptionMgmt

Figure 9. DIS Deployment View: Consuming resource status information

When a DILIGENT Service wants to be notified about events related to a resource profile or status information it contacts its local DIS-HLS Client to discover (via the DIS-DiscoverResource service) the DIS-Broker or DIS-Registry that handles the resource. Then the local DIS-Notification subsystem sends the subscription request to the DIS-Registry (if the service wants to receive notifications about profile changes) or to the DIS-Broker (if the service wants to receive notifications about status information changes) that dispatches the request to the appropriate DIS-IC. When a relevant event occurs, the SubscriptionMgmt subsystem of these services sends the notification to the subscriber by invoking its NotificationHandler (that implements the NotifyInterface interface).

[to be detailed in D1.2.2]

4.4 Significant Classes and components

[to be provided in D1.2.2]

4.4.1 Component: DIS-Registry

The following figure shows how the DVOS registers a resource (and its profile) into the DIS-Registry.

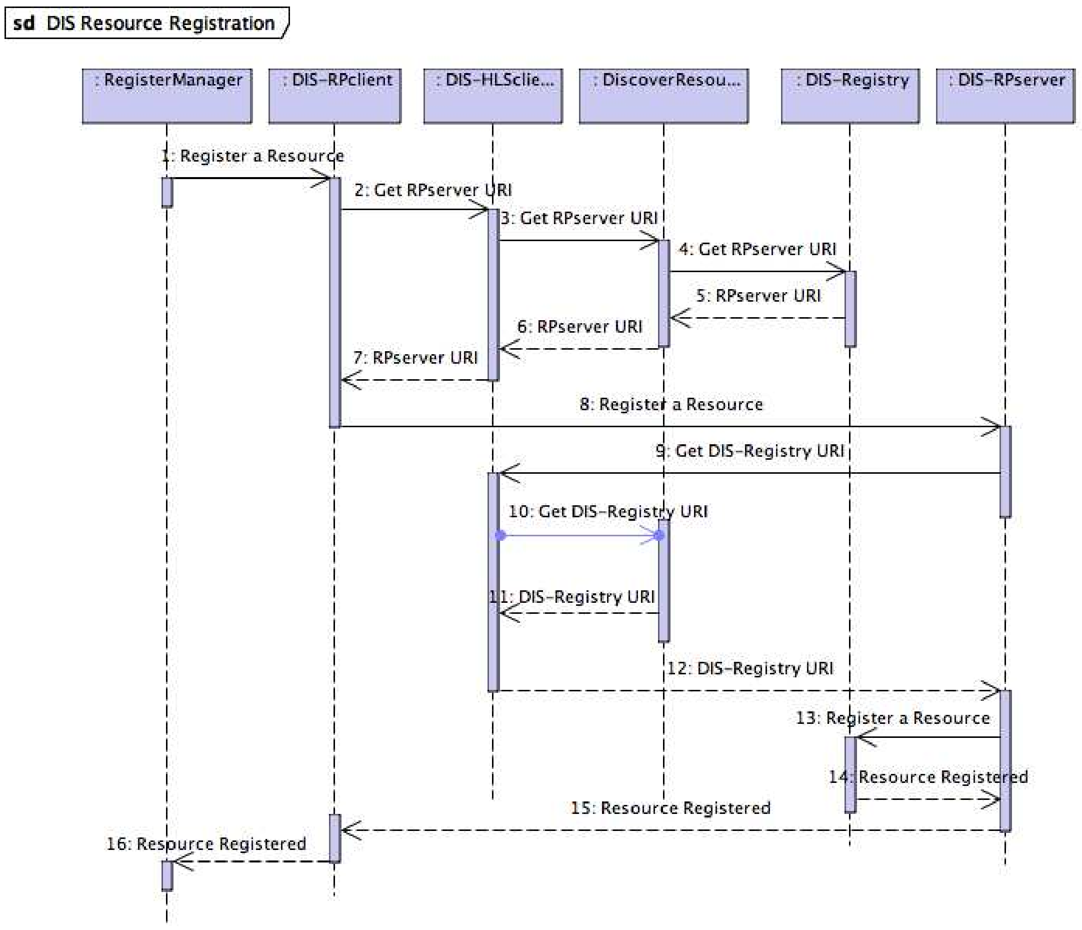


Figure 10. Sequence diagram: DIS resource profile registration

1. A *RegisterManager* contacts the *DIS-RPClient* located on the same DHN in order to register the resource.
2. The *DIS-RPClient* contacts the *DIS-HLSClient* located on the same node in order to get the RP-Server URI.
3. *DIS-HLSClient* uses the discovery component to obtain the *RP-Server* URI. The URI of the *DiscoverResource* is known to the *DIS-HLSClient*.
4. The *DiscoverResource* queries the *DIS-Registry* component and obtain the *RP-Server* URI. The *DiscoverResource* knows the URI of the *DIS-Registry*.
5. The *DIS-Registry* returns the *RP-Server* URI to the *DIS-DiscoverResource*.
6. The *DIS-DiscoverResource* returns the *RP-Server* URI to the *DIS-HLSClient*.
7. The *DIS-HLSClient* returns the *RP-Server* URI to the *DIS-RPClient*.
8. The *DIS-RPClient* registers the resource on the *DIS-RPServer*.

9. The *DIS-RPServer* uses the *DIS-HLSClient* located on the same node in order to get the *DIS-Registry* URI. Note that this *DIS-HLSClient* could not be the same that the *DIS-HLSClient* contacted in step 2 if the *RegisterManager* is located on a different node than the *DIS-RPServer*.
10. The *DIS-HLSClient* uses the *DIS-DiscoverResource* component to obtain the *DIS-Registry* URI.
11. The *DIS-DiscoverResource* returns the *DIS-Registry* URI to the *DIS-HLSClient*.
12. The *DIS-HLSClient* returns the *DIS-Registry* URI to the *DIS-RPServer*.
13. The *DIS-RPServer* registers the resource in the *DIS-Registry*.
14. The *DIS-Registry* confirms the registration.
15. The *DIS-RPServer* notifies the *DIS-RPClient* about successful resource registration.
16. Via the *DIS-RPClient* the successful resource registration communication is delivered to the *RegisterManager*.

[to be detailed in D1.2.2]

4.4.2 Class: WSRFProducer

Figure 11 shows how a WSRF compliant resource producer is created and how it is used to publish information into the DIS using the push model. This sequence is the same (or with very few differences) for all push producers.

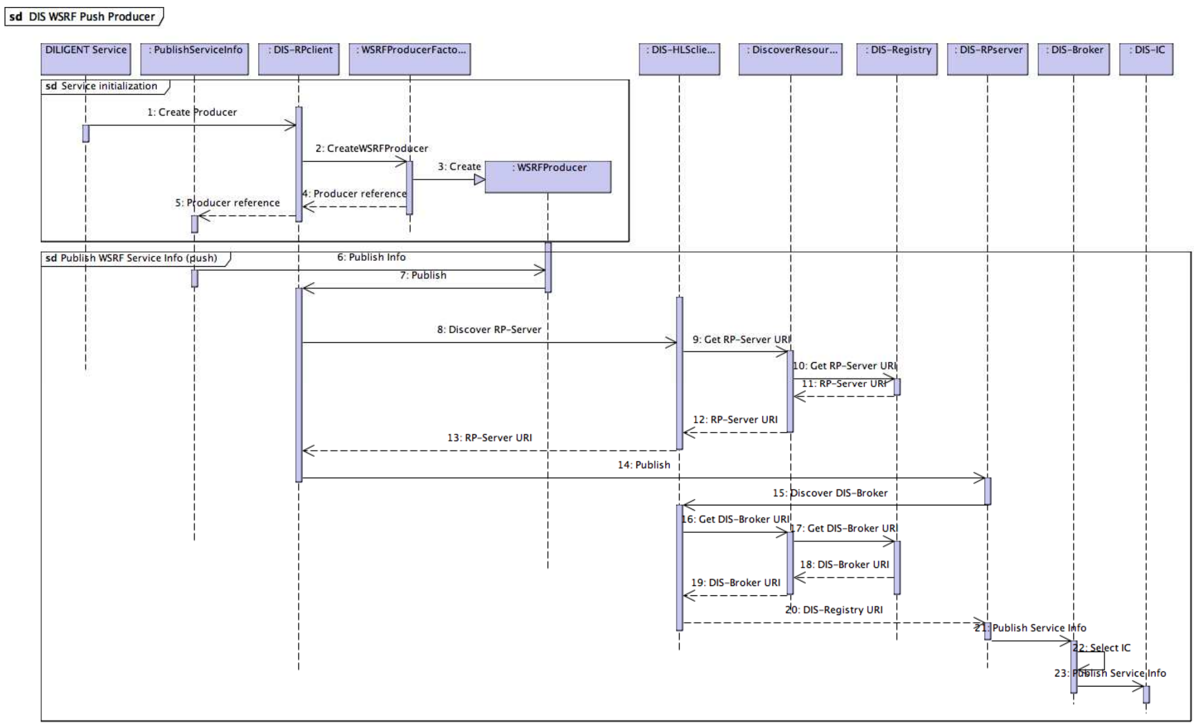


Figure 11. Sequence diagram: publish a WSRF compliant status information (push model)

1. When the resource is deployed it contacts the *DIS-RPClient* located on the same DHN in order to create its producer.
2. The *DIS-RPClient* sends a message to the *Factory* in order to create the appropriate resource producer.
3. The *Factory* creates the producer (*WSRFProducer*)

4. The *Factory* returns the reference to the new producer.
5. The *DIS-RPClient* returns to the DILIGENT service the reference to the new producer. The PublishServiceInfo class of the service uses this reference.
6. When its status information change, the resource decides to publish them by contacting the producer.
7. The resource producer contacts the *DIS-RPClient* located on the same node in order to publish the new status information.
8. The *DIS-RPClient* discovers the *DIS-RPServer* using the *DIS-HLSClient* located on the same node.
9. The *DIS-HLSClient* uses the *DIS-DiscoverResource* component to obtain the *RP-Server* URI. The *DIS-HLSClient* knows the URI of the *DIS-DiscoverResource*.
10. The *DIS-DiscoverResource* queries the *DIS-Registry* component and obtains the *RP-Server* URI. The *DIS-DiscoverResource* knows the URI of the *DIS-Registry*.
11. The *DIS-Registry* returns the *RP-Server* URI to the *DIS-DiscoverResource*.
12. The *DIS-DiscoverResource* returns the *RP-Server* URI to the *DIS-HLSClient*.
13. The *DIS-HLSClient* returns the RP-Server URI to the *DIS-RPClient*.
14. The *DIS-RPClient* publishes the resource information using the *DIS-RPServer*.
15. The *DIS-RPServer* contacts the *DIS-HLSClient* located on the same node in order to get the DIS-Broker URI. Notice that this *DIS-HLSClient* could not be the same that the *DIS-HLSClient* contacted on step 8 if the DILIGENT service is located on a different DHN than the *DIS-RPServer*.
16. The *DIS-HLSClient* uses the *DIS-DiscoverResource* to obtain the DIS-Broker URI.
17. The *DIS-DiscoverResource* queries the registry component and obtains the *DIS-Broker* URI.
18. The *DIS-Registry* returns the DIS-Broker URI to the *DIS-DiscoverResource*.
19. The *DIS-DiscoverResource* returns the *DIS-Broker* URI to the *DIS-HLSClient*.
20. The DIS-HLSClient returns the DIS-Broker URI to the DIS-RPServer.
21. The *DIS-RPServer* sends the new service status information to the *DIS-Broker*.
22. The *DIS-Broker* selects the appropriate information collector to be used.
23. The *DIS-Broker* publishes the service information into the selected DIS-IC.

[to be detailed in D1.2.2]

4.4.3 Class: MonitorProducer

Figure 12 presents what happens if a resource (a DILIGENT service in this case) selects the pull model. In particular, it shows the communication process between the DILIGENT service and the local pull producer (MonitorProducer).

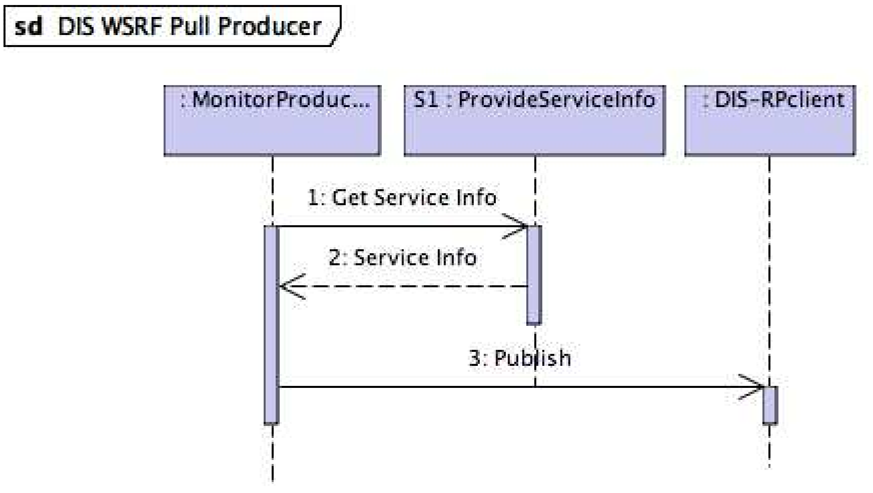


Figure 12. Sequence diagram: publish from a WSRF compliant producer (pull model)

1. The *MonitorProducer* periodically calls the *ProvideServiceInfo* class (which is an implementation of the *InformationProvider* interface presented in the Logical View) in order to update resource information published into the DIS.
2. The resource returns its status information.
3. The producer publishes that information using the *DIS-RPClient* located on the same DHN. After this step, steps from 8 to 23 of Figure 11 are repeated.

[to be detailed in D1.2.2]

4.4.4 Component: DIS-Registry

Figure 13 shows how an update to the resource status information takes place using the pull model for the gLite services. Access to gLite service status is performed through the gLite RGMA Server, i.e. the Service Discovery tool.

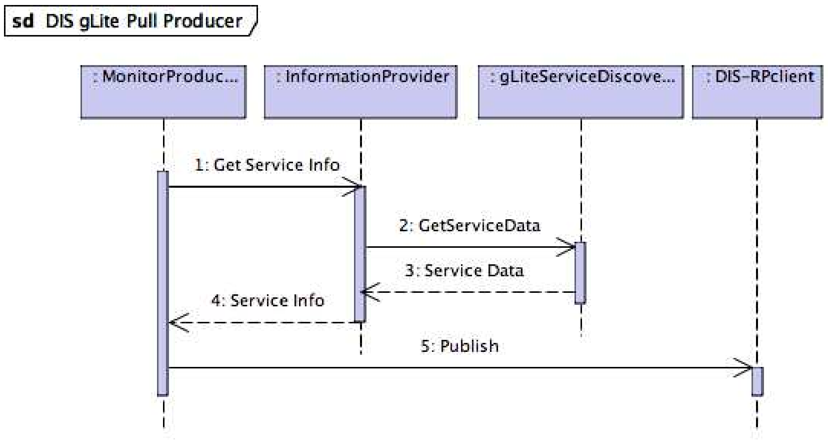


Figure 13. Sequence diagram: provide information status from gLite service (pull model)

1. The *MonitorProducer* periodically calls the *ProvideServiceInfo* class (which is an implementation of the *InformationProvider* interface presented in the Logical View) of the *DIS-R-GMA Client* in order to update resource status information.
2. The *DIS-R-GMA Client* uses the *gLite Service Discovery Tool* to harvest gLite services status.
3. The *gLite Service Discovery Tool* returns the gLite services data.
4. The *DIS-R-GMA Client* returns the gLite services data to the *MonitorProducer*.
5. The *MonitorProducer* publishes that information using the *DIS-RPClient* located on the same node. After this step, steps from 8 to 23 of Figure 11 are repeated.

[to be detailed in D1.2.2]

4.4.5 Component: DIS-IC

[to be provided in D1.2.2]

4.4.6 Components: DIS-Notification/SubscriptionMgmt

Figure 14 shows the notification mechanism that can be used by DILIGENT services to be notified about events related to resource profiles.

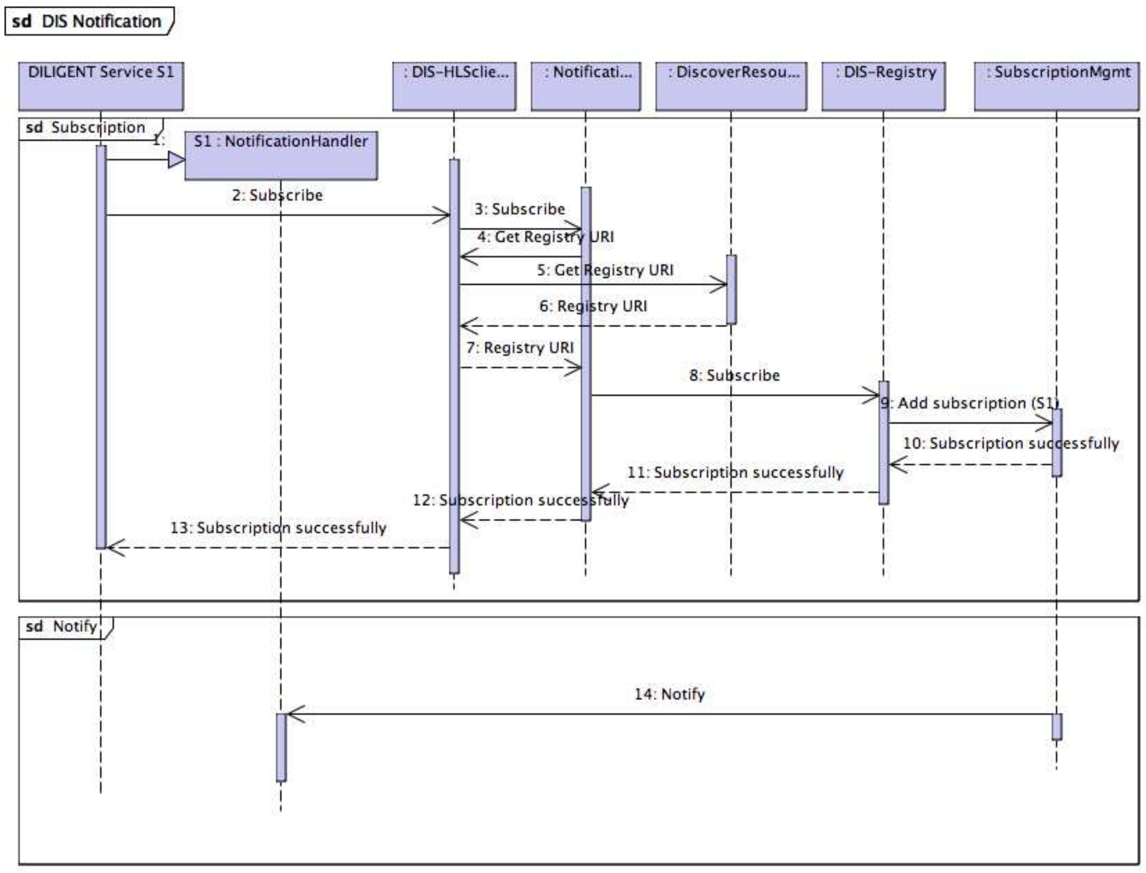


Figure 14. Sequence diagram: Notifications for resource profile changes

1. The *DILIGENT service* creates a *NotificationHandler* to receive notifications.

2. The *DILIGENT service* contacts the local *DIS-HLSClient* to subscribe to events giving the reference to the *NotificationHandler* created.
3. The *DIS-HLSClient* uses its *DIS-Notification* subsystem to manage the subscription.
4. The Notification subsystem uses the *DIS-HLSClient* to retrieve the Registry URI.
5. The *DIS-HLSClient* contacts the *DIS-DiscoverResource* to retrieve the Registry URI. The *DIS-HLSClient* knows the URI of the *DIS-DiscoverResource*.
6. The *DIS-DiscoverResource* knows the URI of the *DIS-Registry* and returns it to the *DIS-HLSClient*.
7. The *DIS-HLSClient* returns the *DIS-Registry* URI.
8. The Notification subsystem subscribes into the *DIS-Registry* the *NotificationHandler* of the DILIGENT service (S1).
9. The *DIS-Registry* adds the handler to its *SubscriptionMgnt*.
10. If the registration is ok the *SubscriptionMgnt* returns success.
11. The *DIS-Registry* returns success.
12. The *DIS-Notification* subsystem returns success.
13. The *DIS-HLSClient* returns success to the *DILIGENT service*.
14. When a relevant event occurs and there are registered handlers for that event, the DILIGENT service *NotificationHandler* is notified.

[to be detailed in D1.2.2]

Figure X shows the notification mechanism that can be used by DILIGENT services to be notified about events related to resource status information.

Figure X
[to be provided in D1.2.2]

4.4.7 Other Components

[to be provided in D1.2.3/D1.2.2]

5 BROKER & MATCHMAKER SERVICE

The Broker & Matchmaker Service (BMS) 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, the BMS job is to propose a set of – possibly ranked[7] - deployment solutions.

5.1 Use-Case View

Deliverable D1.1.1 on Functional Specifications 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 needed for the new Digital Library. To return such a configuration to the Keeper, the BMS performs a matching between package requirements (for each package to be deployed) and the DHNs currently available for the VO.

Normally, the BMS doesn't gather the status of the whole infrastructure while it bases its computation on information supplied by the Diligent Information Service (DIS). The current status of DHNs, periodically fetched from the DIS or notified by it upon any change in DHNs configuration, is stored in the BMS catalogue.

The following diagram includes the 'findOptimalAllocation' functionality plus a number of internal functionalities for the service. Each of them is briefly described below.

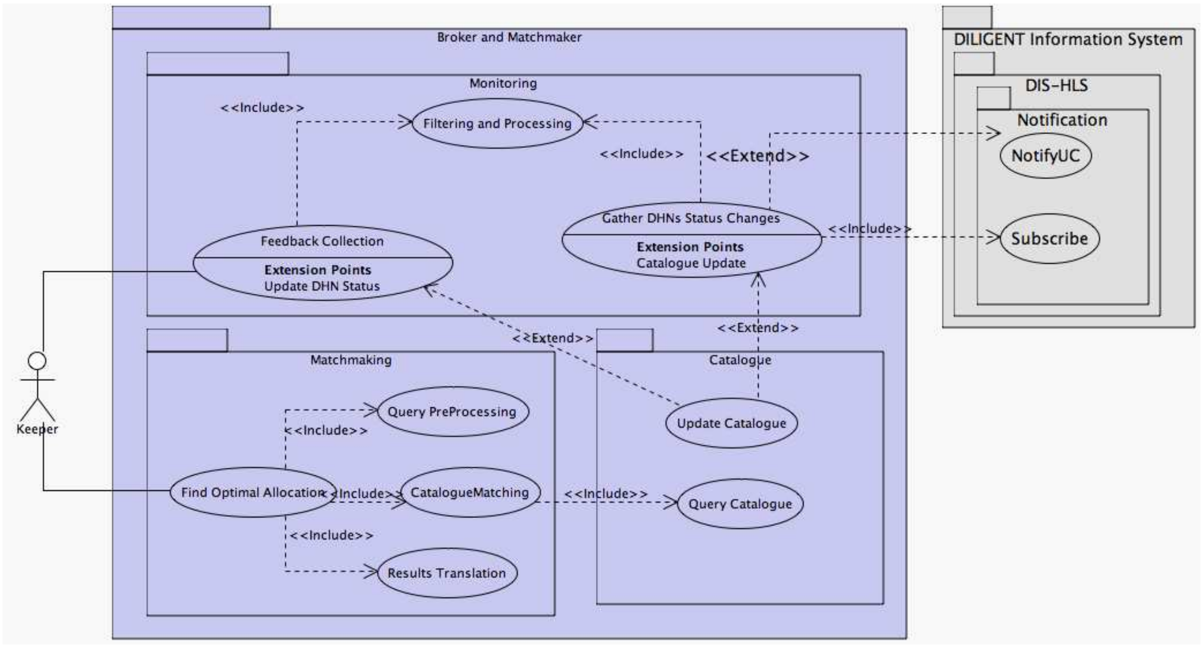


Figure 15. Broker & Matchmaker - Use-Case View

Gather DHN Status Changes

The BMS subscribes itself to the DIS in order to be notified about any changes in DHN availability and status. This functionality is realized through the subscription/notification mechanism provided by the DIS.

[7] Possible ranking measures are those related to the Quality of Services and will be defined accordingly with early system tests.

Filtering and Processing

Notifications are internally filtered and processed. The Filtering phase avoids overload on unneeded information while the Processing phase interprets and translates the notification content into the catalogue-specific language; in this process the BMS could even derive other information from the one supplied by DIS.

Update Catalogue

This use case models the internal storage of received notifications from DIS and received feedbacks from the Keeper after filtering and processing. This information is subsequently exploited for matchmaking against Keeper requests.

Find Optimal Allocation

FindOptimalAllocation is the core functionality of the Broker & Matchmaker Service; it enables diligent services (typically Collective Layer Services, e.g. the Keeper) to discover candidate DHNs in order to deploy packages conforming to the Packages Data Model described in Section 6.4.2. This functionality is based on three further internal functionalities described hereafter.

Query PreProcessing

The language the BMS provides to the Keeper allows to describe requirements on hosting nodes and dependencies between packages; nevertheless matchmaking algorithms expect the query to be expressed in a way that is more suitable for processing. This functionality deals with the parsing, translation and optimization of the request before its submission for matchmaking.

[query language to be detailed in D1.2.2]

Catalogue Matching

This functionality matches the request with the set of DHNs currently modelled in the internal catalogue. The collection of – possibly ranked – acceptable deployment solutions is returned to the caller.

Results Translation

Presents results produced by the matchmaking algorithm using an agreed language for exporting results.

[result language to be detailed in D1.2.2]

Feedback Collection

Even if a configuration matches all the requirements made by a requester, a deployment action, made by the Keeper based on this configuration, may lead to a failure. This happens for several reasons, i.e. changes on the DHNs between the moment the BMS identifies a configuration and the effective deployment action making some resources actually unavailable on the node. This functionality models the gathering of failure notifications from the Keeper in order to force refresh of DHNs status, to rank down the reliability of a DHN, to notify DHN Resource Manager about potential node problems, etc.

Query Catalogue

[to be provided in D1.2.2]

5.2 Logical View

BMS architecture is composed of three main packages:

- a catalogue package maintaining an up-to-date picture of the DHNs status and topology;

- a matchmaker package which is responsible for matching the service request with the catalogue content and providing the requester with a set of possible deployment solutions.
- a monitoring package which is in charge of keeping the BMS catalogue up-to-date with respect to notifications received from the DIS and feedbacks from the Keeper.

No Graphical User Interface is provided by BMS as this service is not meant to interact directly with any human user.

[to be detailed in D1.2.2]

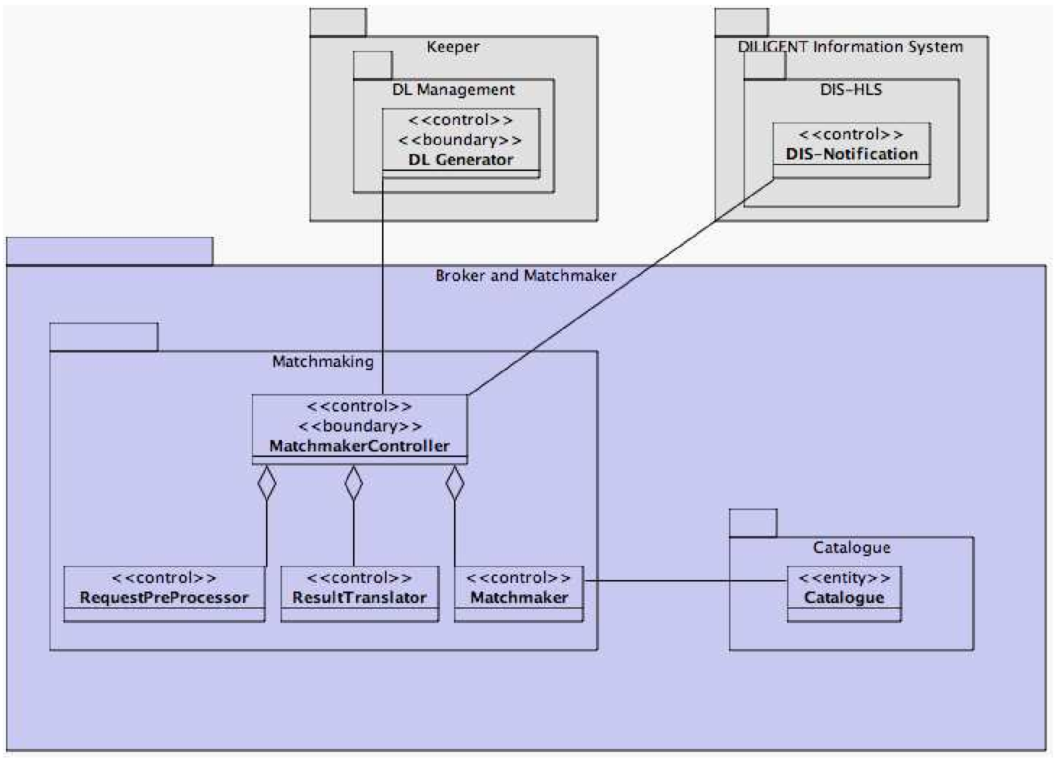


Figure 16. Broker & Matchmaker - Logical View

Catalogue package

The Catalogue package is in charge of maintaining the most up-to-date status of Diligent Hosting Nodes.

As a first set of properties being maintained, the generic host model defined within the GLUE Schema Specification[8] is adopted. The following picture provides an overview of Generic Host Model defined by the GLUE schema specification. This basic model will be extended and enriched to support matchmaking requests. [to be detailed in D1.2.2]

[8] http://infnforge.cnaf.infn.it/projects/glueinfomodel/

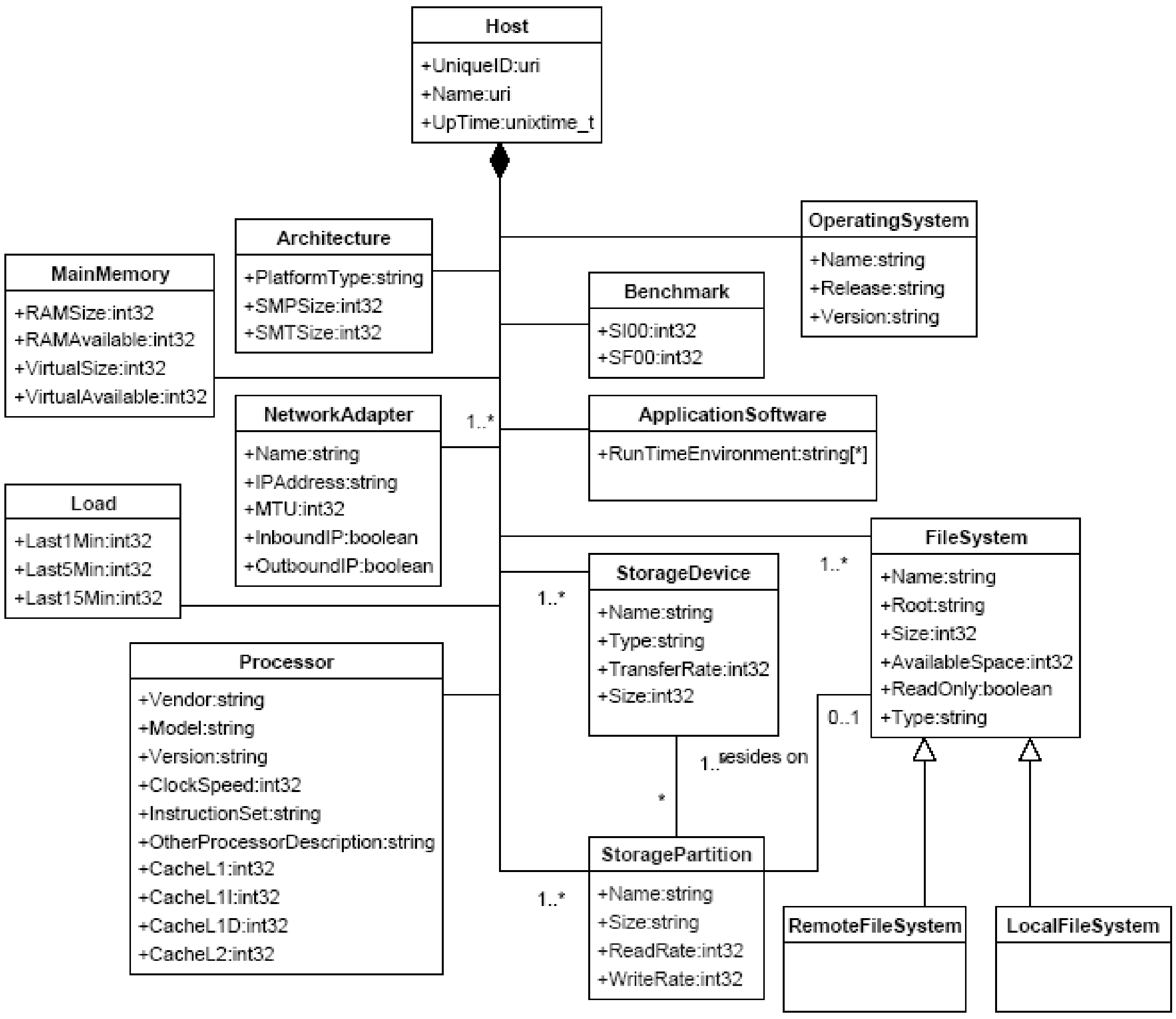


Figure 17. Generic Host model from GLUE Schema

According to the initial design of the Keeper Service (see Section 6) a matchmaking request can express a preference for a DHN equipped with a specified set of already-deployed packages. On the other hand, the GLUE Schema modelling a Generic Host doesn't cope with this kind of information, which is clearly needed.

At this preliminary design stage, we're planning to extend the above Generic Host Model by enriching the Host class with the set of Packages currently installed on it. The definition of the Package class is the one defined in Section 3.

[to be detailed in D1.2.2]

Matchmaking package

This package provides the core functionalities of the BMS service. It actually interacts with external services by accepting requests and presenting possible configurations. Four main classes compose it:

- MatchmakerController is the entry point for the Broker & Matchmaker service and manages the whole internal process;
- RequestPreProcessor realizes the Query PreProcessing functionality transforming an incoming request to be suitable for matchmaking;

- ResultTranslator is in charge of translating results of the matchmaking process to an XML representation to be returned to the requester. The response is further enriched with details not related to matchmaking. 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);
- Matchmaker is the main class of the package, implements a matching algorithm and is responsible for determining possible deployment solutions.

Hereafter a brief overview of the kind of matchmaking requests and responses being handled by the Broker & Matchmaker Service is given.

A matchmaking request is basically composed of two parts:

- A list of packages being deployed and a set of requirements on the DHN according to the GLUE schema host model (e.g. available memory, processor speed, etc.).
- A list of dependencies between packages. Such dependencies intend to force the hosting of two packages on the same DHN or to force the presence of a package in the deployment solutions proposed. With regard to this kind of dependency, several alternatives/possibilities are initially considered:
    a. A dependency can be set over an already-deployed package or over a package in the deployment set.
    b. A dependency can be set over any deployment of package (no matter where it is) or with reference to a particular deployed package – this is the same as specifying a particular DHN for a given package.
    c. A dependency can be given a priority – e.g. 'mandatory', 'desirable', 'optional', etc.

The output of matchmaking is composed by a set of – possibly ranked[9] – alternative deployment solutions. Within each solution, each package is associated with a DHN reference to be deployed on. Optional packages may not be given a DHN to use if no appropriate node is found; furthermore, if the request allows reuse of already-deployed packages, references to them can be returned.

[to be detailed in D1.2.2]

Monitoring Package

The Monitoring package provides a unique entry point for the information about the DHN's status.

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 also deals with gathering of deployment failure feedbacks received by the Keeper, required for ranking tuning.

[to be detailed in D1.2.2]

5.3 Deployment View

From a deployment point of view, matchmaking and monitoring classes are glued together as a single WSRF-compliant service (MM in the diagram below) and deployed on a single

[9] See note 1

DHN. Thus these classes are registered in Diligent as a single WSRFService package (see the Diligent Resources Data Model picture in Section 3).

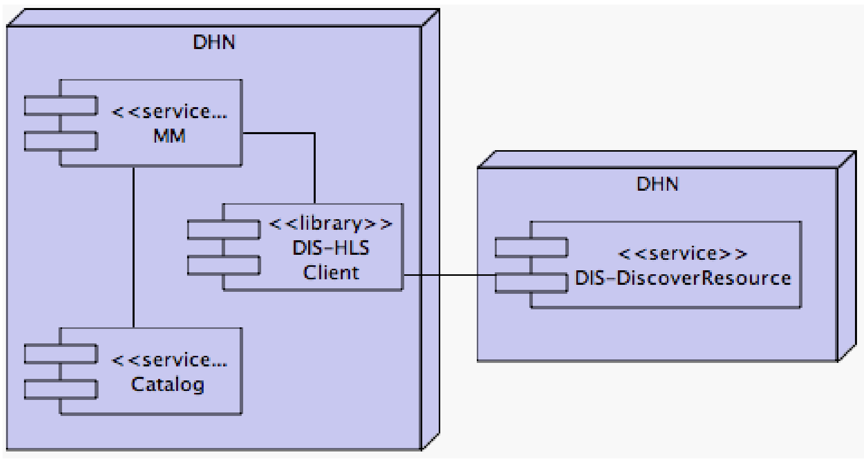


Figure 18. Broker & Matchmaker - Deployment View

Deployment of multiple, independent instances of the BMS can also be considered in order to provide availability and scalability; anyway, the matchmaking being exploited mostly during the deployment of new Digital Libraries, a heavy usage of BMS is not expected frequently.

[to be detailed in D1.2.2]

5.4 Significant Classes and Components

[to be provided in D1.2.2]

6 KEEPER SERVICE

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"

Thus, the Keeper is responsible for the creation of a new VDL by instantiating its resources and archives (or their web service interfaces) and by creating its users. Moreover, it must guarantee the overall set of functionalities of the VDL at any time by dynamically reallocating resources/archives and periodically checking their status.

6.1 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 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 to be able to become a DILIGENT Hosting Node (in short, DHN), i.e. a node able to host DILIGENT services. What this means in detail is explained in the following sections.

Moreover the huge part of the work is performed in a background manner since, 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 specified by the DL Designer.

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

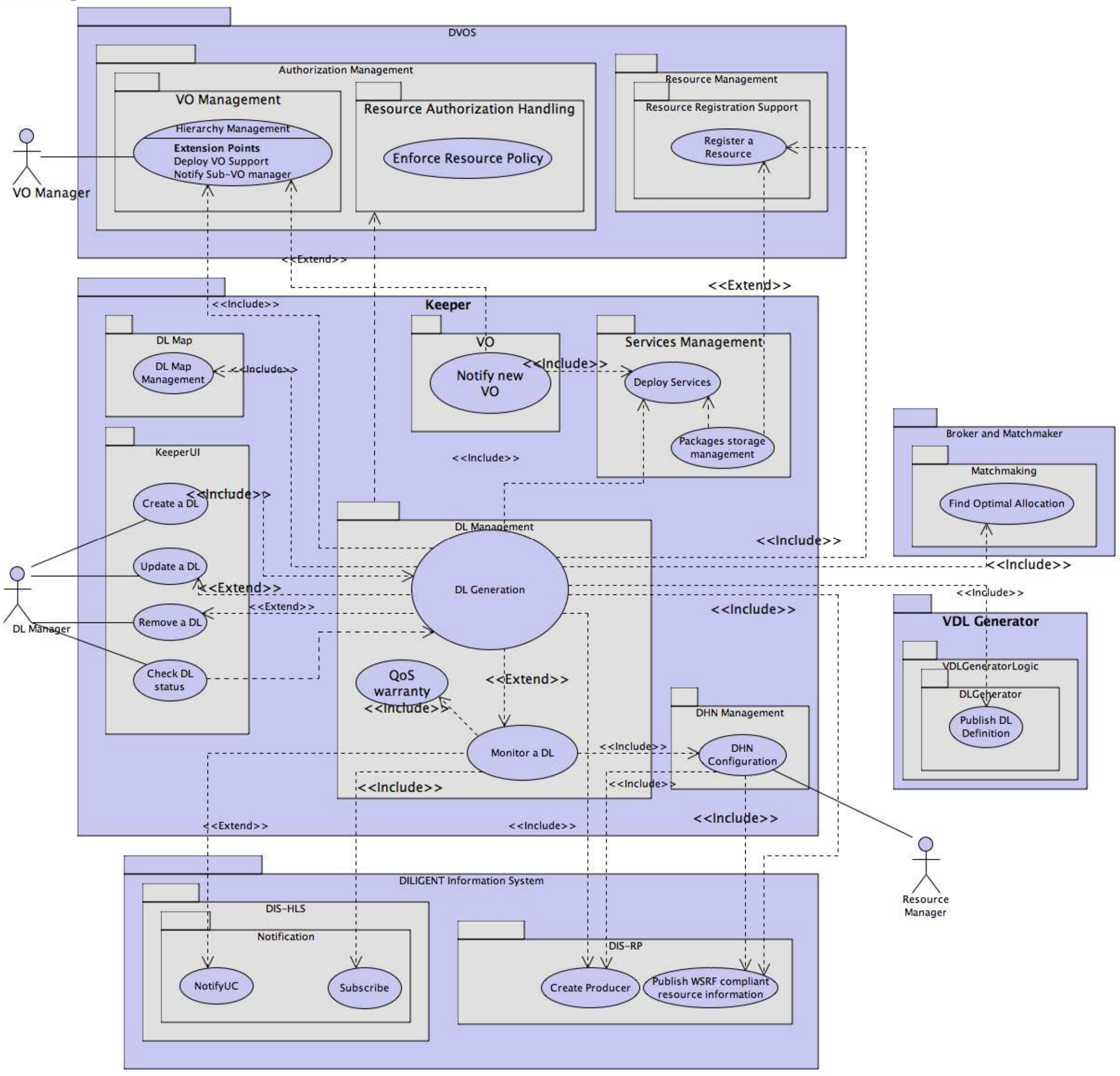


Figure 19. Keeper Service – Use-Case View

The sub-packages inside the main Keeper package identify a preliminary decomposition of the service structure that is used as basis to build the software architecture. Hereafter we present a brief overview of each of them.

KeeperUI package

This package groups use-cases that are directly accessed from the DL Manager, i.e. the graphical user interface of the service.

- **Create a DL** – After the 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 or distribution or notify the DL Designer about a problem that occurs 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.

DL Management package

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

- **DL Generation** – This use-case 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 will compose a new DL.
- **Monitor a DL** – After its creation, a DL must be maintained and monitored in a consistent way. This monitor periodically checks the set of resources that form a DL. If a critical fault occurs in a DL Resource, it tries to recover it by shutting down the resource and by creating a new one on a different DHN. If this doesn't solve the problem or this is not allowed/possible, it notifies the DL Manager about the problem, who decides about subsequent actions.
- **QoS warranty** – This use-case 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:

- from the Keeper point of view, to manage the group of services that form the DL and
- from the other services point of view, to create an application context in a distributed environment.

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

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

Service Deployment package

In order to create new service instances we need to have two major additional functionalities: one that handles the software to be deployed and another one that physically installs the service on a DHN. These use-cases are grouped in the Service Deployment package.

- **Packages storage management** – A service is usually composed by one or more software packages. These packages are uploaded through the Resource Registration Support functionality provided by the DVOS service 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

Finally, we need a component 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:

- o the manual DHN configuration sets by the Resource Manager (as DHN owner)
- o 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 spring 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 **Notify New VO** functionality models this behaviour of the Keeper.

[to be detailed in D1.2.2]

6.2 Logical View

By analyzing the use cases of the use-case view, the logical decomposition of the Keeper Service depicted in Figure 20 results. In this logical vision of the Keeper service, each sub-package presented in the use-case view is expanded in its logical classes and relationships among these classes are also shown. The only package that is not directly reported is the VO package. Its functionalities are included in the VO Deployer, which is a class that is a part of the DL Management, due most of its logic (the deployment of packages) is shared with that class.

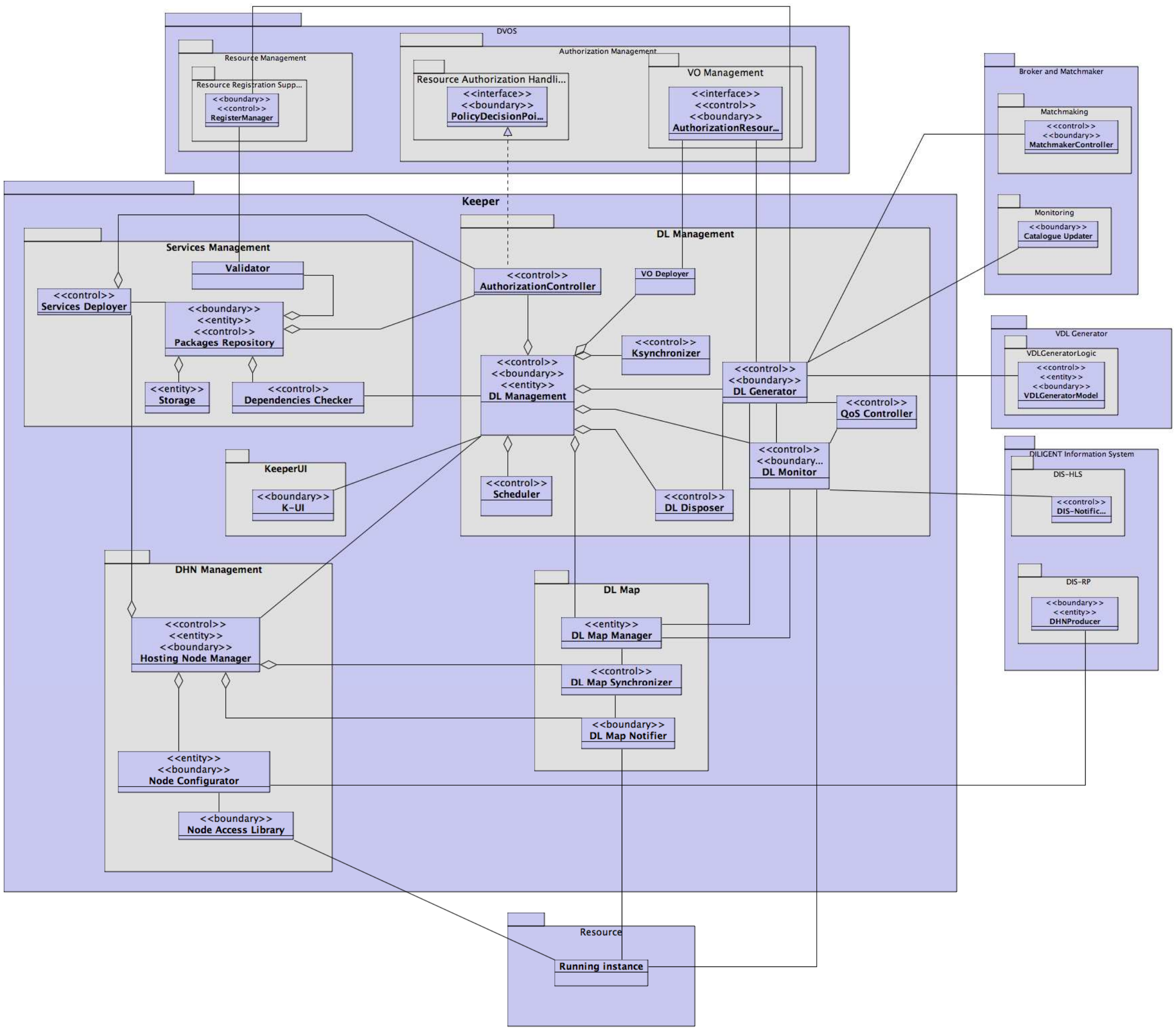

DVOS
Resource Management
Resource Registration Supp...
<<boundary>> <<control>> RegisterManager
Authorization Management
Resource Authorization Handli...
<<interface>> <<boundary>> PolicyDecisionPoi...
VO Management
<<interface>> <<control>> <<boundary>> AuthorizationResour...
Broker and Matchmaker
Matchmaking
<<control>> <<boundary>> MatchmakerController
Monitoring
<<boundary>> Catalogue Updater
Keeper
Services Management
Validator
<<control>> Services Deployer
<<boundary>> <<entity>> <<control>> Packages Repository
<<entity>> Storage
<<control>> Dependencies Checker
DL Management
<<control>> AuthorizationController
VO Deployer
<<control>> Ksynchronizer
<<control>> <<boundary>> <<entity>> DL Management
<<control>> <<boundary>> DL Generator
<<control>> QoS Controller
<<control>> <<boundary... DL Monitor
<<control>> Scheduler
<<control>> DL Disposer
KeeperUI
<<boundary>> K-UI
VDL Generator
VDLGeneratorLogic
<<control>> <<entity>> <<boundary>> VDLGeneratorModel
DILIGENT Information System
DIS-HLS
<<control>> DIS-Notific...
DIS-RP
<<boundary>> <<entity>> DHNProducer
DHN Management
<<control>> <<entity>> <<boundary>> Hosting Node Manager
<<entity>> <<boundary>> Node Configurator
<<boundary>> Node Access Library
DL Map
<<entity>> DL Map Manager
<<control>> DL Map Synchronizer
<<boundary>> DL Map Notifier
Resource
Running instance

Figure 20. Keeper Service - Logical View

A very 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 5.4 during the description of the related component.

[to be detailed in D1.2.2]

DL Management package

The DL Management includes classes that manage the creation, monitoring and disposal of service instances. These classes work in conjunction with the DHN Management and Service Management package to perform the deployment process.

Regarding the interactions with other DILIGENT services, they deal with:

- 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, to deploy mandatory packages for a new VO
- Generic DILIGENT Service – to query its status and accept notifications about its faults

[to be detailed in D1.2.2]

DL Map package

These classes maintain and disseminate the map of the DL. They should be used by any DILIGENT service to discover information (mainly the location) about other services. Details about the usage of these classes are provided in Section 6.4.3.

[to be detailed in D1.2.2]

KeeperUI package

The KeeperUI interacts directly 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 query the DL Management package classes in order to obtain a report of the DL status at any time.

[to be detailed in D1.2.2]

DHN Management package

These classes are involved in the creation of a new DL. Their role is to accept the instructions of the DL Management classes and to start the deployment process using the Service Management classes.

[to be detailed in D1.2.2]

Service Management package

Its role is to store the packages of software and to manage their deployment process on DHN nodes following the instruction of the DHN Management classes. It also interacts with the DIS service in order to be notified about the availability of new packages registered by resource owners.

[to be detailed in D1.2.2]

6.3 Deployment View

Starting from the logical view depicted in Figure 20, four main classes have been selected to become major Keeper components that can be hosted in different networked locations:

- Packages 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 6.4.4).

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 Packages Repository, ii) the DL Management, and iii) the Hosting Node Manager.

[to be detailed in D1.2.2]

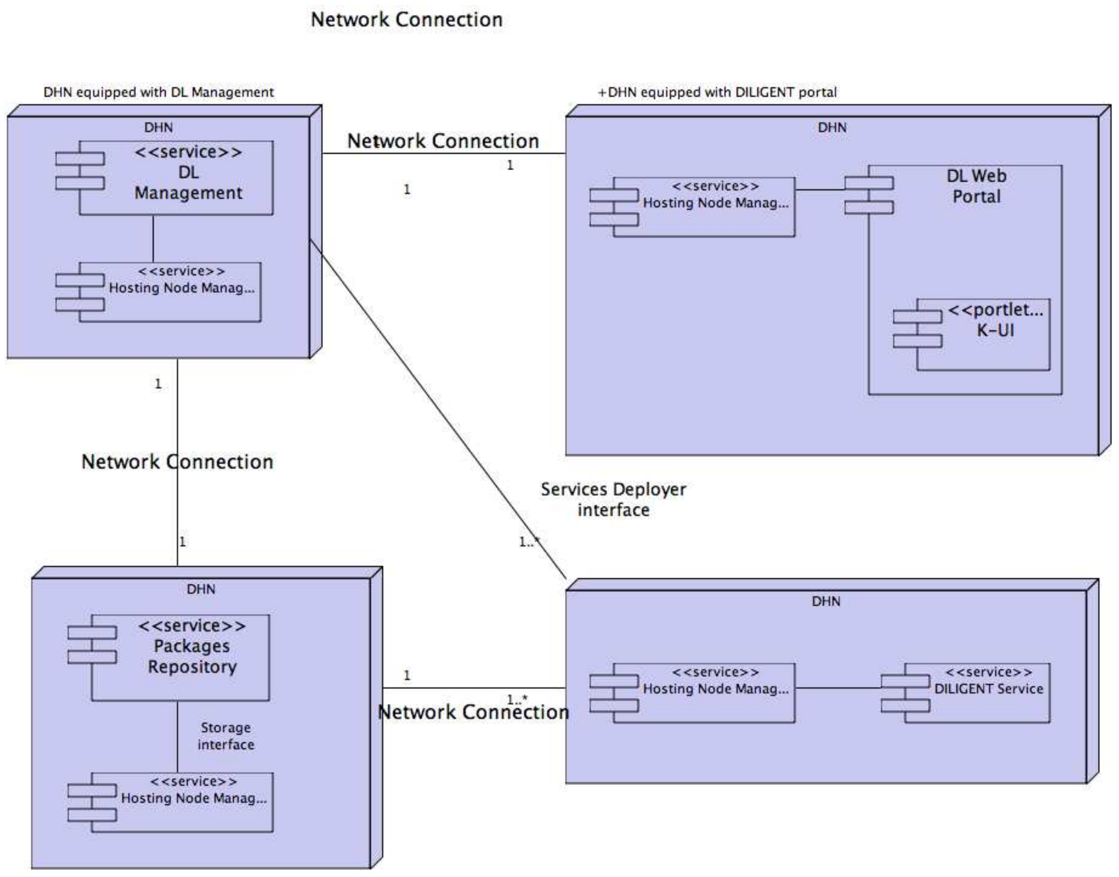

Network Connection
DHN equipped with DL Management
+DHN equipped with DILIGENT portal
DHN
<<service>> DL Management
<<service>> Hosting Node Manag...
Network Connection
1
1
DHN
<<service>> Hosting Node Manag...
DL Web Portal
<<portlet... K-UI
1
Network Connection
Services Deployer interface
1
1..*
DHN
<<service>> Packages Repository
Storage interface
<<service>> Hosting Node Manag...
1
1..*
Network Connection
DHN
<<service>> Hosting Node Manag...
<<service>> DILIGENT Service

Figure 21. Keeper Service - Deployment View

The DL Management is the central component in charge of the creation of a new DL (due to its importance it will be deployed in a replicated way, in figure it is reported one of these replicas); the Hosting Node Manager is the component that must reside on each networked node able to host DILIGENT services and it is in charge of the deployment of new packages on the node; the Packages Repository is a packages manager that handles all the software installable on demand. The K-UI portlet is the graphical user interface integrated in the DL portal that permits the execution of some administrative tasks and to view a report of the DL status.

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

6.4 Significant Classes and Components

6.4.1 Component: DL Management

DL Management is the component in charge to build and manage a DL. It is instantiated each time a new VO is created. It is composed by several modules (also known as “subsystems” in UML terminology). Figure 22 depicts the distribution of these modules and their interactions with other Keeper components on other nodes.

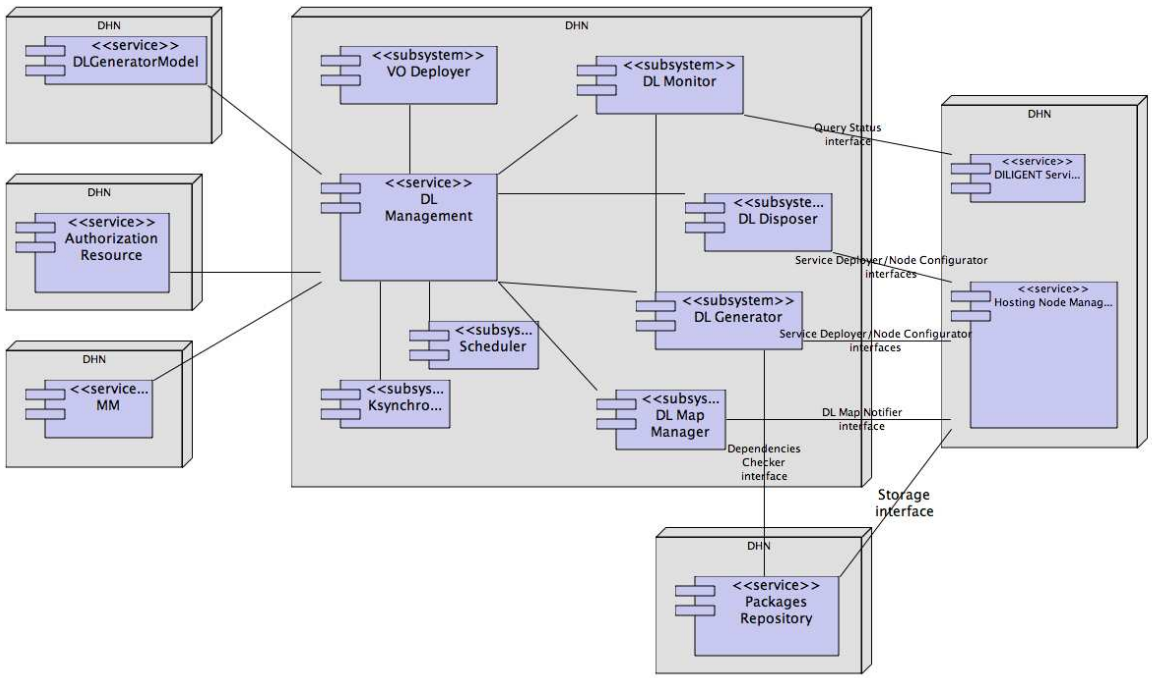

DHN
<<service>> DLGeneratorModel
DHN
<<subsystem>> VO Deployer
<<subsystem>> DL Monitor
DHN
<<service>> Authorization Resource
<<service>> DL Management
DHN
<<service... MM
<<subsyste... DL Disposer
<<subsys... Scheduler
<<subsys... Ksynchro...
<<subsystem>> DL Generator
<<subsys... DL Map Manager
Query Status interface
Service Deployer/Node Configurator interfaces
Service Deployer/Node Configurator interfaces
DL Map Notifier interface
Dependencies Checker interface
Storage interface
DHN
<<service>> DILIGENT Servi...
<<service>> Hosting Node Manag...
DHN
<<service>> Packages Repository

Figure 22. DL Management Node - Deployment Diagram

The following picture shows how the DL Management is involved in the process of creation of a new DL starting from a VO named A.

[to be provided in D1.2.2]

DL Generator subsystem

A new DL Generator is instantiated each time a new DL must be created. The DL Generator elaborates the VDL definition criteria, interacts with the Packages Repository and builds the list of physical resources 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 in order to obtain the DHN nodes on which deploy these resources. Finally, it must be able (and have rights to do this) to join pre-existing resources (e.g. archives, collections) with the rising DL.

The outcomes of these 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

[to be detailed in D1.2.2]

Expected interactions with other services

- The VDLGeneratorModel subsystem of the VDLGenerator service must pass to DL Generator the VDL definition criteria upon which the new VDL will be created. This "document" contains:
    - o list of Resources (Archives and Services)
    - o the configuration of each resource
    - o a special resource and configuration related to the Web Portal
    - o list of invited users/groups
- Interact with the Packages Repository in order to retrieve the dependencies among the different packages
- Interact with the Broker & Matchmaker service 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 responds with a list of possible solutions for the deployment. Each solution is composed by:
    - o a list of packages
    - o for each package: a DHN that satisfies the requirements associated with the package at the package registration time
- Interact with the Dynamic VO Support in order to create a new sub-VO and add the DL Resources and DL Users to it
- Interact with *Service Deployer* subsystem on the DHNs to load/configure/activate services and components on the node
- Interact with the DVOS service in order to register a new "Running instance" (see Section 3) as a DILIGENT Resource after each deployment

[to be detailed in D1.2.2]

DL Disposer subsystem

This module disposes a DL when the DL Manager decides to remove it. It should de-allocate (or deactivate) on the DHN nodes all the packages related to the DL if they are not shared with other DLs. Each Running Instance must expose a method to call in order to make persistent its status. Since this operation is service-dependent, the DL Dispose is not aware about what happens when it calls this method.

Expected interactions with other services/components

Interact with Service Deployment Module on the DHNs to unload/deactivate services and components on the node

[to be detailed in D1.2.2]

DL Monitor subsystem

The DL Monitor is the module in charge of monitoring the whole DL. It assumes that each DILIGENT service exposes an interface that can be queried in order to investigate the service status. When a service instance fails the DL Monitor asks the DL Generator to deploy a new instance of that service or, if it is not possible (because the service was not deployed 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).

Moreover, it must guarantee that a DL maintains the level of QoS requested by the DL Designer.

[to be detailed in D1.2.2]

Scheduler subsystem

[to be defined in D1.2.2]

Map Manager subsystem

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

[to be detailed in D1.2.2]

Expected interactions with other services/components

Interact with the DL Map Synchronizer module of the Hosting Node Manager in order to send notifications about DL Map changes

[to be detailed in D1.2.2]

KSynchronizer subsystem

A DL Management instance is a vital component of the entire DILIGENT application. Therefore, DL Management needs to be replicated. This module assures the synchronization among different DL Management instances.

[to be detailed in D1.2.2]

VO Deployer subsystem

The VODeployer 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 Packages that are labeled as "VOMandatory". Of course, these deployment operations are performed in conjunction with the Broker & Matchmaker service that selects the target DHNs.

The following picture shows how the Keeper service is involved in the process of creation of a new VO named A starting from the DILIGENT VO.

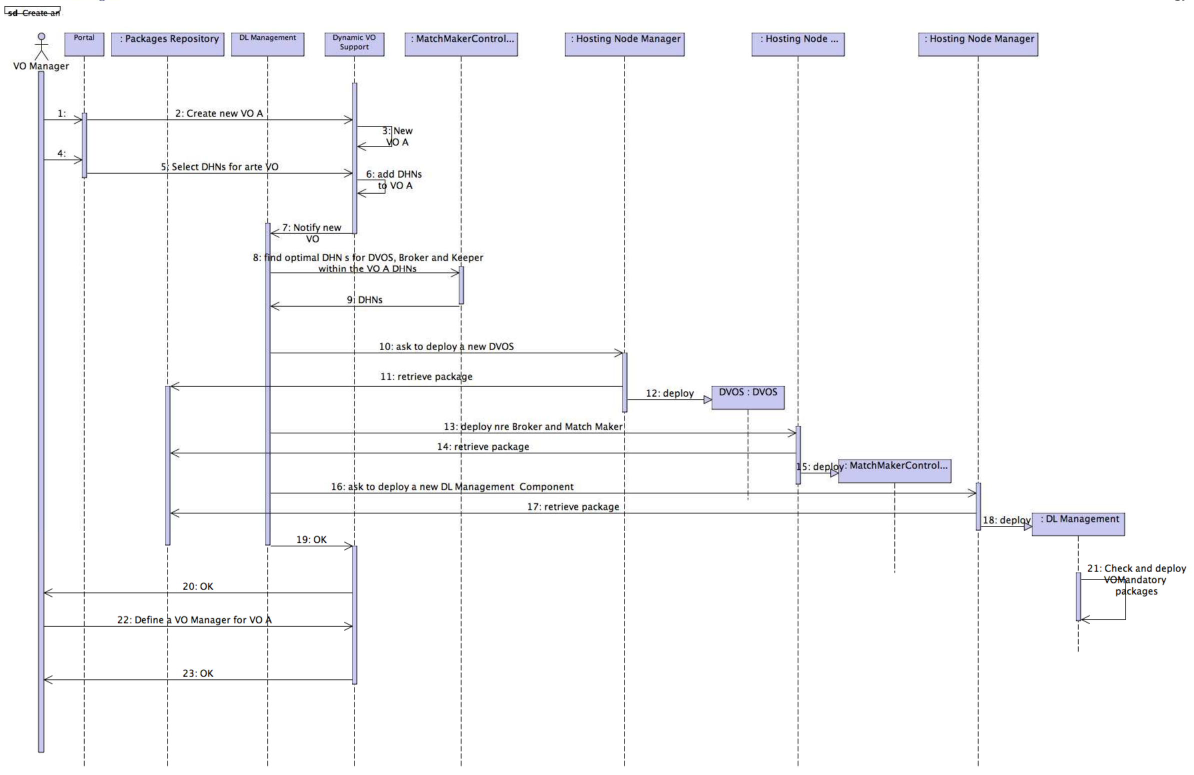


Figure 23. Keeper Service - The creation of a new VO

[to be detailed in D1.2.2]

6.4.2 Component: Packages Repository

Usually, a package is defined as a software component prepared for integration into a suitable computer system. In the DILIGENT context, a package is a "piece of software" able to be deployed on a DHN. The "preparation" of a package must support any of its usage phases: DHN environment preparation, installation, customization, activation, update, replacement or removal. The package owner that registers the package as a DILIGENT Resource in the Dynamic VO Support Service must do this preparation. This service uploads the new package into the Packages Repository that validates the package (its conformity to the Package Model presented below) and returns a positive/negative response. If the validation is ok, the DVOS is in charge of registering the new resource in the DIS.

Figure 24 depicts the Package Model adopted. A new package uploaded in the Packages Repository is validated against this logical model.

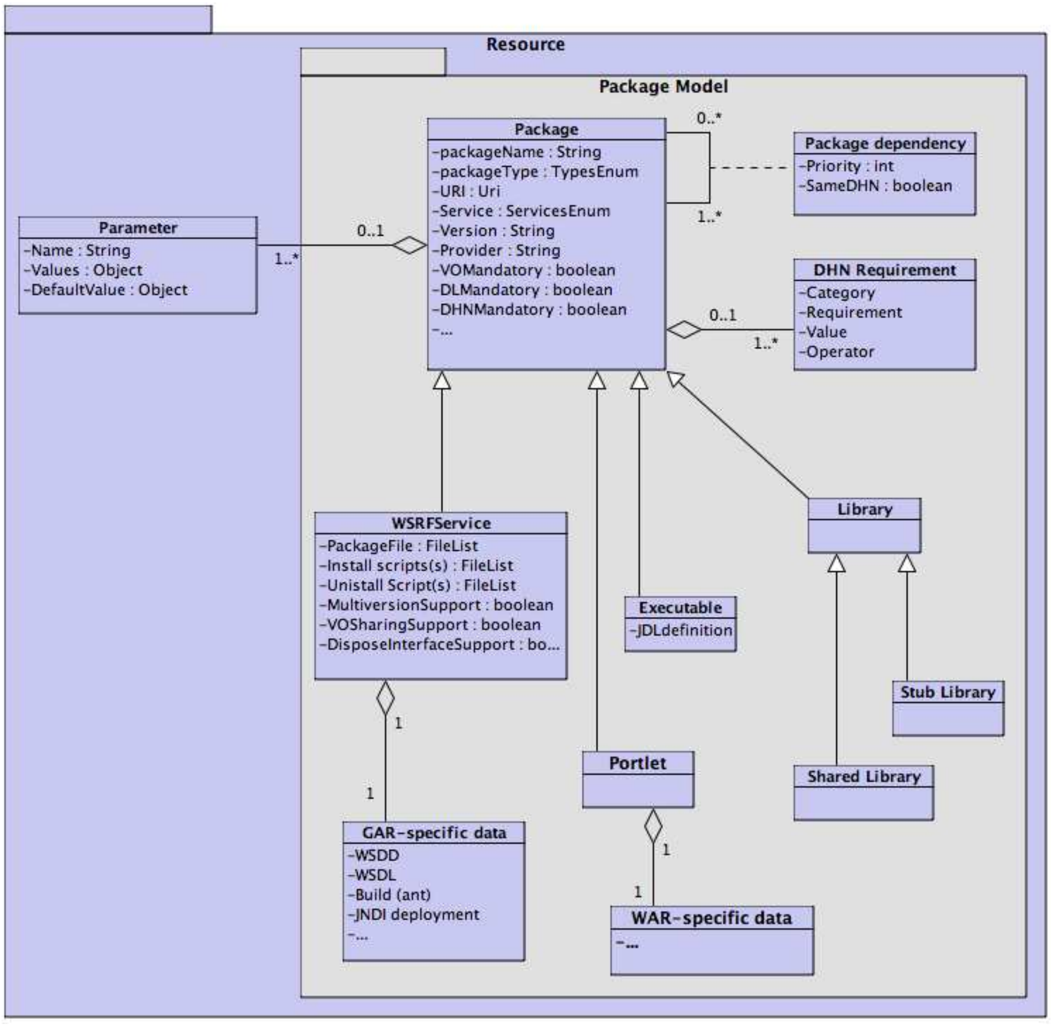


Figure 24. Package Model

Of course, not all resources are uploaded into the Packages 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 (Executables)
- Shared/Stub libraries
- Portlets

The entry point to access one of these resources is maintained by the DIS-Registry. When the Keeper needs to retrieve a package, it first performs a discovery operation on the DIS-Registry and then it accesses the resource in the Repository.

Due 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 distribution of the different components of the Packages Repository is depicted in the following figure.

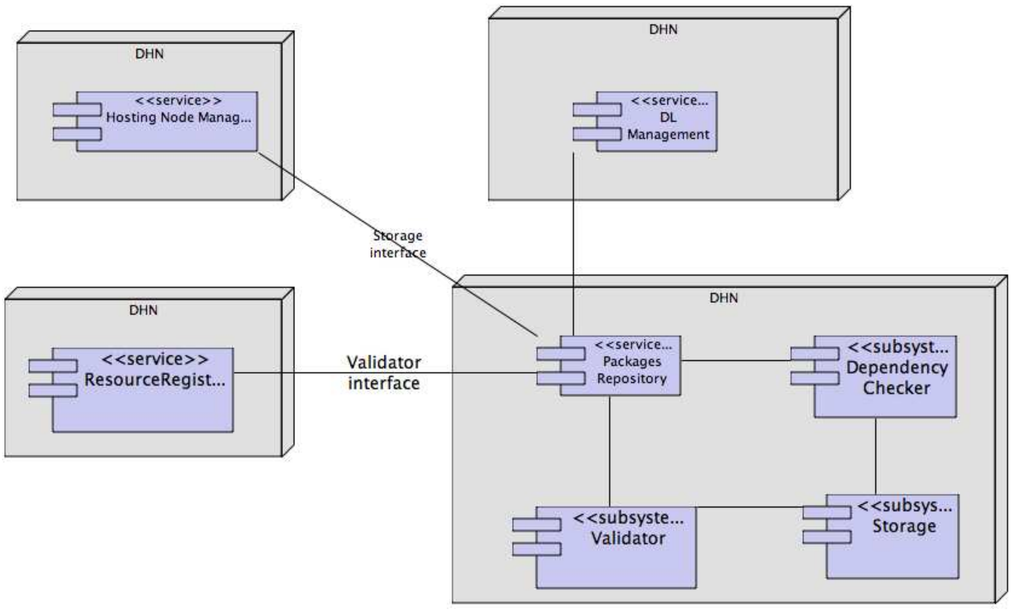


Figure 25. Packages Repository Node - Deployment Diagram

Subsystems

[to be provided in D1.2.2]

Expected interactions with other services/components

- the DVOS notifies and uploads new software packages in the Packages Repository; the Packages Repository validates (or not) each package and returns the result to the DVOS that registers the package in the DIS if it is compliant with the Packages Model presented in Section 3.1.

[to be detailed in D1.2.2]

Assumptions

Each new package must be conformed to the data model presented in Section 3.1.

[to be detailed in D1.2.2]

6.4.3 Component: Hosting Node Manager

A DHN is simply a node able to host DILIGENT services and related components. In order to become a DHN, a node (that is already part of the DILIGENT infrastructure) must have installed the following software:

- Java WS Core
- Hosting Node Manager

At the time of writing this deliverable, there is not a final decision if this software is dynamically deployed on the node or if it must be manually installed. Since this decision does not affect the design of the Keeper (or not influence what we have designed so far), as first step we can decide to install it manually and then, when our knowledge of the system and the grid technologies will be improved we can study a possible solution to dynamically deploy this software in an automatic way.

Java WS Core

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 host services. These solutions are usually referred to with 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 by which we are deploying the services is the Java WS Core and developed by the Globus Alliance. This container provides a complete implementation of the WSRF [15] 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.

Hosting Node Manager

The Hosting Node Manager is the minimal DILIGENT software that must be present on a DHN. Any other software can be dynamically deployed starting from this component. The HNM is the manager of the node on which it is hosted. The node management involves the following tasks:

- Deploy new DILIGENT Services on the node
- Maintain and expose the node configuration to the hosted services
- Exchange data with the DL Management Component
- Push DHN configuration to the DIS

Figure 26 presents the deployment diagram for this Keeper component.

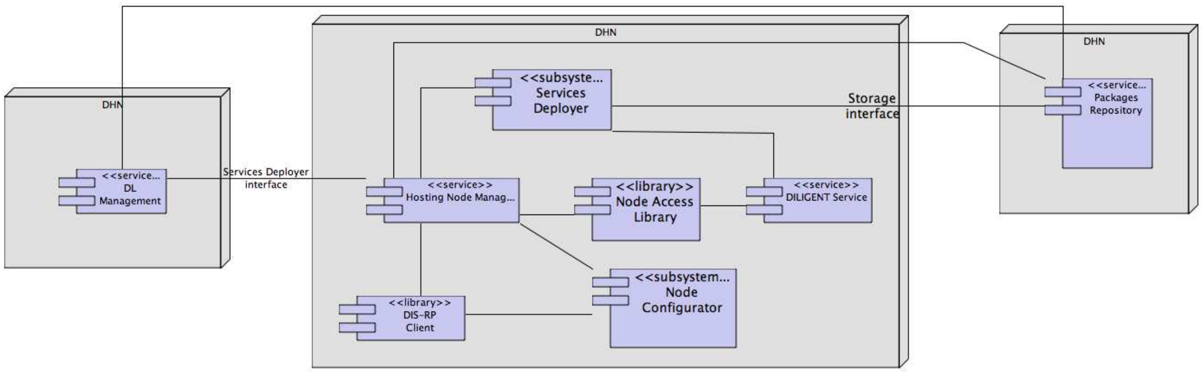

DHN
DHN
DHN
<<service... DL Management
Services Deployer interface
<<subsyste... Services Deployer
Storage interface
<<service... Packages Repository
<<service>> Hosting Node Manag...
<<library>> Node Access Library
<<service>> DILIGENT Service
<<subsystem... Node Configurator
<<library>> DIS-RP Client

Figure 26. DHN Node - Deployment Diagram

In the following sections each task of the Hosting Node Manager is described in details.

6.4.3.1 Deploy new DILIGENT Services on a node

The Deployment

The deployment operations of a service on a DHN are strictly related to the hosting environment that hosts the service itself. 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 WS plain.

[to be detailed in D1.2.2]

[to be detailed in D1.2.3]

Building and deploying Grid services

In order to be deployed in the Java WS Core, the Grid services (uploaded in the Packages Repository as WSRFService packages) must be distributed using the GAR packaging format. A GAR (Grid Archive) file is a single file that 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 build and deploy GAR files, the build tool Ant developed by the Apache Foundation is adopted. The Java WS Core provides a number of predefined Ant tasks that can be used to perform common operations with a very low additional effort necessary to customize.

[to be detailed in D1.2.2]

Service Deployer subsystem

This module offers the interface to use in order to manage services on a node. The interface provides the following primitives:

- (un-)load active service (accept/remove software packages)
- (un-)set active service
- (re-)configure active service
- (de-)activate active service

From the Resource Model perspective, the role of the Service Deployer is to transform one or more packages in a running instance of a service hosted in the DHNs where the Service Deployer is running. The following picture is a subset of the Resource Model that shows the different classes involved in this stage.

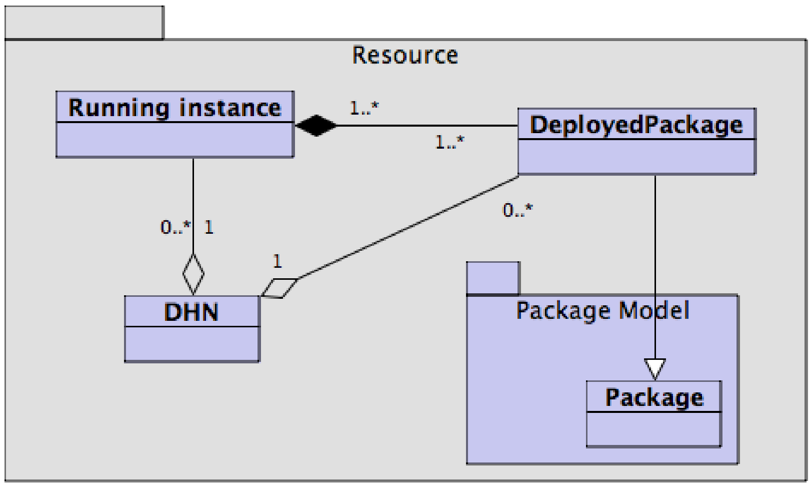


Figure 27. Service Deployer Model

Interact with the Packages Repository to retrieve software packages to deploy.
Interact with the local Java WS Core container to install new packages.

[to be detailed in D1.2.2]

Expected interactions with other services
[to be detailed in D1.2.2]

6.4.3.2 Maintain and expose the node configuration to the hosted services

This area groups the following subsystems:

DL Map Notifier subsystem

This module provides a mechanism to subscribe to notifications about any changes in the current DL Map. Each service on the node uses this module to be updated about the services with which they interact.

[to be detailed in D1.2.2]

DL Map Synchronizer subsystem

[to be provided in D1.2.2]

Node Configurator subsystem

On each DHN there is a set of basic configuration data; these data can be passed by the DL Management Component or can depend on the local node configuration. They include at least:

- the URL of the DIS-DiscoverResorce service of the DIS
- the URL of the local AuthorizationResource of the DVOS
- the installed software (third parties software)
- the hardware configuration
- …

This information is partially provided by the Resource Manager that joins the DHN with the DILIGENT infrastructure and partially provided by the DL Management component at run time.

[to be detailed in D1.2.2]

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.

[to be detailed in D1.2.2]

6.4.3.3 Exchange data with the DL Management Component

[to be detailed in D1.2.2]

DL Map Synchronizer subsystem

This module synchronizes the local map of the DL with the one maintained by the DL Management Component. It uses the classes provided with that implement WS-Notification to subscribe and receive information about changes.

[to be detailed in D1.2.2]

6.4.3.4 Push DHN configuration to the DIS

[to be provided in D1.2.2]

6.4.4 Component: K-UI

[to be provided in D1.2.2]

7 DYNAMIC VO SUPPORT SERVICE

DILIGENT Service Oriented Architecture, based on Web Services model, allows 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 specifications [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 paragraph 3.7 of the test-bed functional specifications [1]. Moreover, managing identity mapping among DILIGENT and gLite domains, DVOS enables DILIGENT services to join existing gLite-based infrastructures.

The DVOS area covers functionalities related to Resource Management (paragraph 4.3 of D1.1.1), to VO Management (4.4 of D1.1.1), to Users and Group Management (4.5 of D1.1.1) and to Notification Management (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 specifications [1]. 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

4.4 Authorization Management

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

- 4.4.18 Get User’s VO-Resources

4.6 Notification Management

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

4.5 User and Group Management

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

4.3 Resource Management

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

Due to the heterogeneity of the DVOS functionalities this service is decomposed in five sub-packages. Diagram in Figure 28 summarizes this decomposition showing internal dependencies among packages. The internal structure of each functional package is explained in following sections.

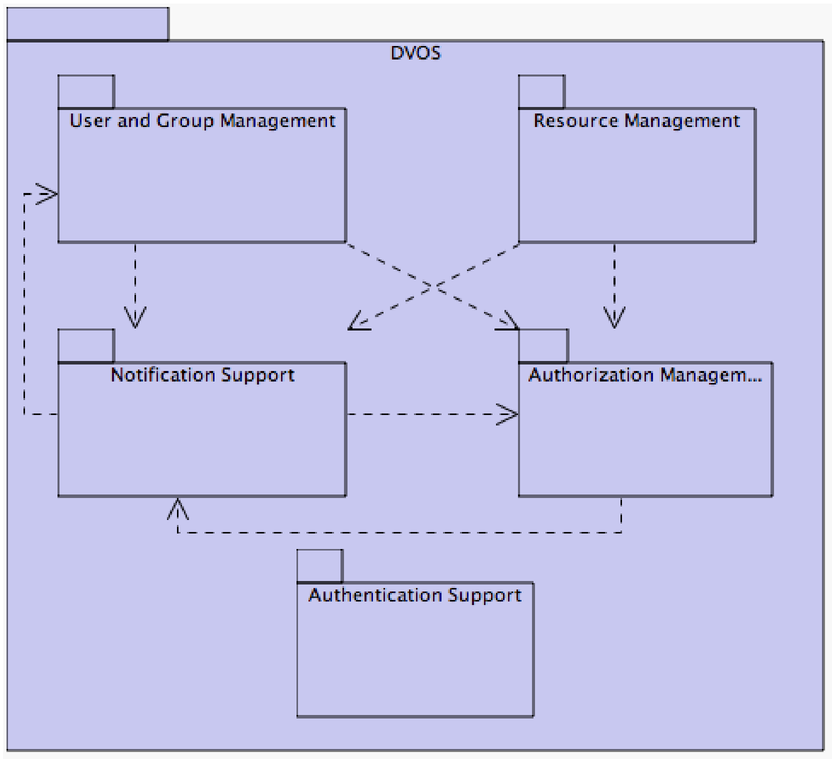


Figure 28. DVOS packages

[VO Data Model to be added here in D1.2.2]

[to be detailed in D1.2.2]

7.1 Authentication Support

The DILIGENT Authentication model is based on Grid Security Infrastructure (GSI)[10] standards. This implies the adoption of a Public Key Infrastructure (PKI) and the usage of X.509 End Entity Certificates (EEC) [6] released by a Certification Authority (CA) in order to authenticate DILIGENT users. GSI standards also support 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 assignment)
- Service Container instances (static assignment)

The static assignment of EECs to Web Services instances and Service Containers implies that a copy of credentials used by these entities must be available in the machine hosting the Container or the Web Service instance.

[10] 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.

Authentication support deals with user identity and credentials management issues. Main functionalities provided by this area are listed below. A brief description of each one 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. It 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. 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 DILIGENT User credentials. 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, as it is required in order to use existing gLite infrastructures. Some DILIGENT users or entire DILIGENT VOs could share the same gLite account in order to access an existing gLite infrastructure. Moreover the mapping is not static and it 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 29. This view does not include functionalities related with 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 29).

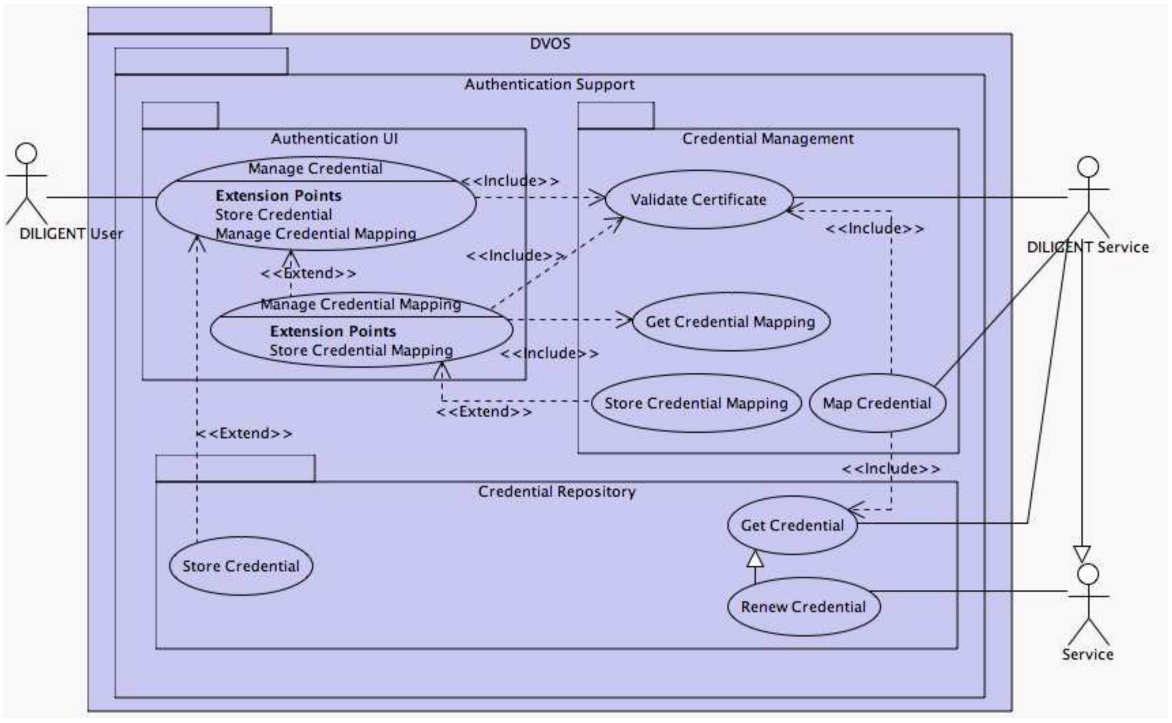


Figure 29. Authentication Support – Use Case View

Actors involved in authentication scenario are:

- *DILIGENT Users*, responsible to store a middle-term copy of their credentials in the repository and to manage mapping between DILIGENT and gLite credentials.
- *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.

[to be detailed in D1.2.2]

A brief description of packages in Figure 29 is provided below.

Credential Management package

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

[to be detailed in D1.2.2]

- **Validate Certificate**

  [to be provided in D1.2.2]
- **Map Credential**

  [to be provided in D1.2.2]
- **Get Credential Mapping**

  [to be provided in D1.2.2]
- **Store Credential Mapping**

  [to be provided in D1.2.2]

Authentication UI package

Through the Authentication UI each *DILIGENT User* can store its own credentials in the repository and manage its own credentials mapping.

VO Managers can also use this interface to assign gLite credentials to entire DILIGENT VOs. These credentials are used during the *Map Credential* operation and enable access to existing gLite-based grid infrastructures.

[to be detailed in D1.2.2]

- **Manage Credential**

  [to be provided in D.1.2.2]

- **Manage Credential Mapping**

  [to be provided in D.1.2.2]

Credential Repository package

This package contains functionalities related with credential repository. This component enables services (DILIGENT and gLite) to retrieve user credential and to renew it during execution.

[to be detailed in D1.2.2]

- **Store Credential**

  This operation allows users to store a middle-term copy of their DILIGENT or gLite certificates in the repository. The lifetime of the stored copy as well as the password needed to retrieve the short-term copy of credentials are set by the user.

  [to be detailed in D.1.2.2]

- **Get Credential**

  In order to get a short-term copy of user credentials the username and password previously set by the user are needed.

  [to be detailed in D.1.2.2]

- **Renew Credential**

  In order to renew short-term user credentials an already valid copy of credentials is needed.

  [to be detailed in D.1.2.2]

7.1.2 Logical View

Main entities involved in certificate management are:

- One (or more) trusted CA, which is 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 *Renew Credential* operation.
- A Mapping Service, which is in charge to *Store Credential Mapping* and to perform the *Map Credential* operation.

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

The analysis of the use cases in Figure 29 leads to the following logical view. Figure 30 contains main classes belonging to the Authentication Support package.

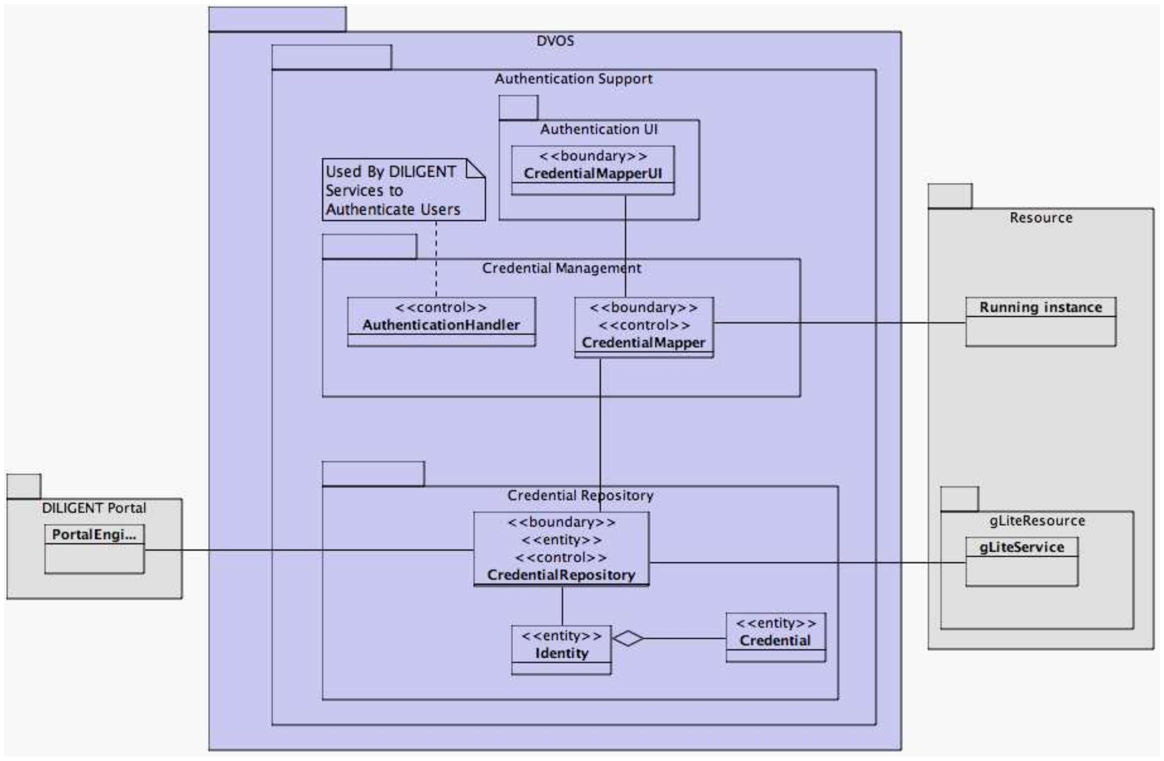

DVOS
Authentication Support
Authentication UI
<<boundary>> CredentialMapperUI
Used By DILIGENT Services to Authenticate Users
Credential Management
<<control>> AuthenticationHandler
<<boundary>> <<control>> CredentialMapper
Resource
Running instance
DILIGENT Portal
PortalEngi...
Credential Repository
<<boundary>> <<entity>> <<control>> CredentialRepository
<<entity>> Identity
<<entity>> Credential
gLiteResource
gLiteService

Figure 30. Authentication Support – Logical View

CredentialMapperUI

CredentialMapperUI enables *DILIGENT Users* to manage credentials mapping associated to their identity.

[to be detailed in D1.2.2]

CredentialRepository

This is 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.

[to be detailed in D1.2.2]

Identity

This entity class models the *DILIGENT User* account in the credential repository.

[to be detailed in D1.2.2]

Credential

This class models a single set of credentials associated with a *DILIGENT User*. Each DILIGENT User can upload in the repository more than one credential, but only one of them can be associated with the *DILIGENT User* identity. Other credentials will be exploited during access to existing grid infrastructures.

[to be detailed in D1.2.2]

CredentialMapper

This component maintains mapping between *DILIGENT* and gLite identities and it provides other DILIGENT Services with on-demand credentials delegation.

[to be detailed in D1.2.2]

AuthenticationHandler

This handler class allows DILIGENT services to authenticate incoming requests through the validation of the PC attached to service requests. This class is provided by the underlying GSI implementation and the Java WS Core automatically performs the PC validation during service request handling.

[to be detailed in D1.2.2]

7.1.3 Deployment View

[to be provided in D1.2.2]

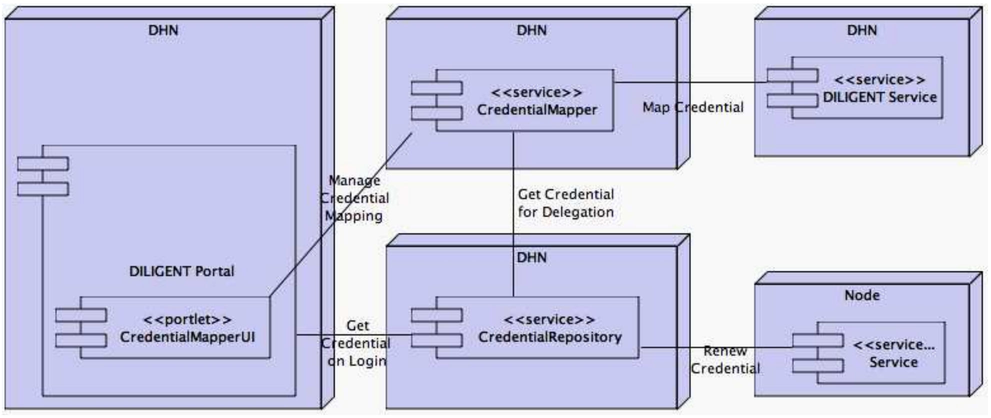

DHN
DIGILENT Portal
<<portlet>> CredentialMapperUI
Manage Credential Mapping
Get Credential on Login
DHN
<<service>> CredentialMapper
Get Credential for Delegation
Map Credential
DHN
<<service>> DILIGENT Service
DHN
<<service>> CredentialRepository
Renew Credential
Node
<<service... Service

Figure 31. Authentication Support – Deployment View

7.1.4 Significant Classes and components

7.1.4.1 Component: CredentialRepository

The following diagram shows a very high level view of the DILIGENT authentication scenario.

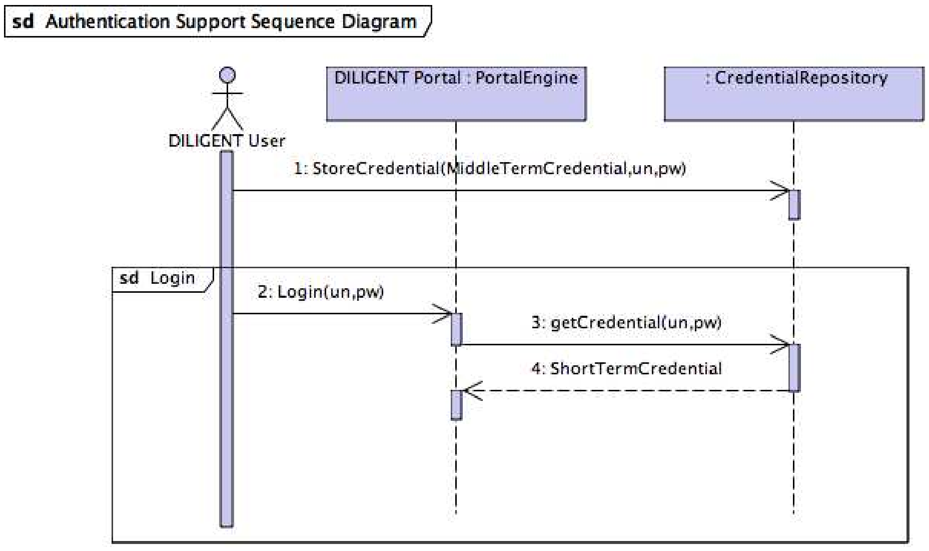

sd Authentication Support Sequence Diagram
DILIGENT User
DILIGENT Portal : PortalEngine
: CredentialRepository
1: StoreCredential(MiddleTermCredential,un,pw)
sd Login
2: Login(un,pw)
3: getCredential(un,pw)
4: ShortTermCredential

Figure 32. Authentication Support – Login sequence diagram

Step 1 shows the credentials storage operation performed by the DILIGENT User from a machine where its EEC and private key are available. The user sets the lifetime of middle-term credentials. As shown in Figure 32 this operation can be performed without the support of the DILIGENT portal.

Steps from 2 to 4 show the user login and the retrieval of short term credentials performed by the DILIGENT portal. Users can perform this operation by every machine equipped with a browser supporting HTTPS features (in order to securely send username and password to the DILGIENT portal).

[to be detailed in D1.2.2]

7.1.4.2 Component: CredentialMapper

The following diagram shows the access to a gLite Service from the DILIGENT infrastructure.

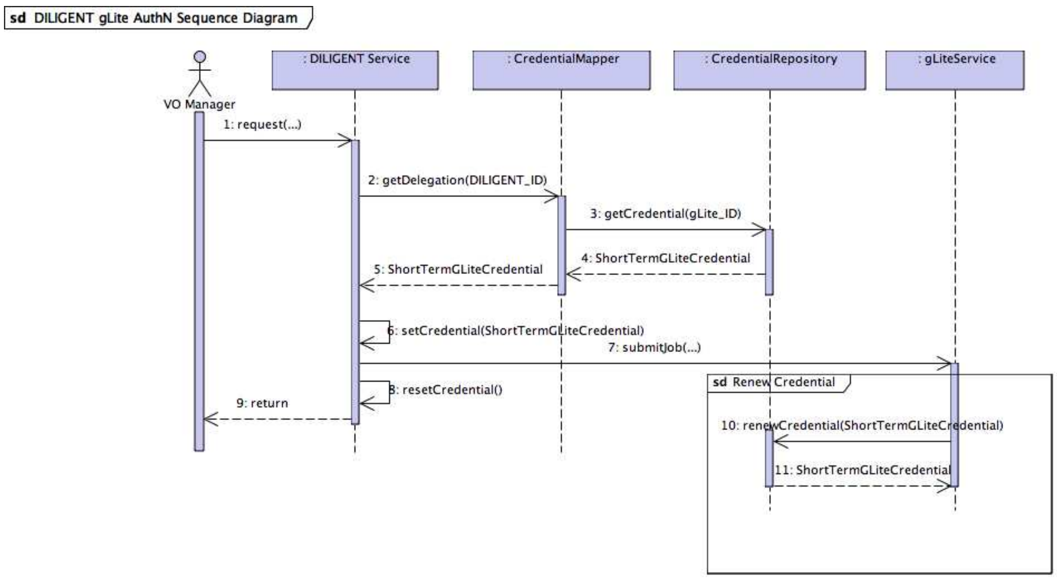

sd DILIGENT gLite AuthN Sequence Diagram
: DILIGENT Service
: CredentialMapper
: CredentialRepository
: gLiteService
VO Manager
1: request(...)
2: getDelegation(DILIGENT_ID)
3: getCredential(gLite_ID)
4: ShortTermGLiteCredential
5: ShortTermGLiteCredential
6: setCredential(ShortTermGLiteCredential)
7: submitJob(...)
8: resetCredential()
9: return
sd Renew Credential
10: renewCredential(ShortTermGLiteCredential)
11: ShortTermGLiteCredential

Figure 33. Authentication Support – gLite access sequence diagram

The DILIGENT service contacts the *CredentialMapper* service providing the certificate of the DILIGENT User in order to obtain the corresponding gLite credentials (step 2). GLite credentials for the user are obtained from the *CredentialRepository* (step 3 and 4) and forwarded to the DILIGENT Service (step 5). The service set the gLite credentials to use (step 6) and access to the gLite service, e.g. to submit a job (step 7). After job submission credentials are reset to the original ones and the DILIGENT service returns to the caller.

Steps 10 and 11 show the operation of credentials renewal during job execution performed using the short-term certificate provided with the job submission.

[to be detailed in D1.2.2]

7.2 Authorization Management

This package deals with VO-level Authorization issues. The DILIGENT VO-level Authorization model originates from the VO model as described in the paragraph 3.7 of the D1.1.1 Functional Specifications [1].

The X.509 standard that defines the Proxy Certificates format allows Proxy Certificate (PC) to carry authorization information as certificate extensions [6]. These extensions are included in the PC during its creation. This functionality can allow authorization mechanisms at every site to resolve locally authorization issues, without the need to contact a remote authorization service. Nevertheless this model has two constrains: i) authorization information contained in the Proxy Certificate are static and can't be updated, ii) the authorization extensions have a lifetime that could be strongly different than the Proxy certificate one.

In the DILIGENT infrastructure, where resources are dynamically created and destroyed, a dynamic authorization model is needed. DILIGENT services must be enabled to allow access based on up-to-date authorization information.

The DILIGENT authorization model relies on the underlying Java WS Core authorization framework. It enables DILIGENT services to separate security issues from service specific behavior. This is achieved through the use of 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 the service. The authorization chain can be configured through an XML file named "Security Descriptor"[11].

A particular authorization handler will be provided to DILIGENT services by the Authorization Management package in order to enforce VO-level authorization policies defined by VO Managers. DILIGENT services can also enforce finer grained authorization policies defining and configuring service-specific authorization handlers.

[to be detailed in D1.2.2]

7.2.1 Use Case View

The analysis of the DILIGENT VO model (see paragraph 3.7 of [1]) points out the need for a structure capable to maintain 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 services. 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.

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

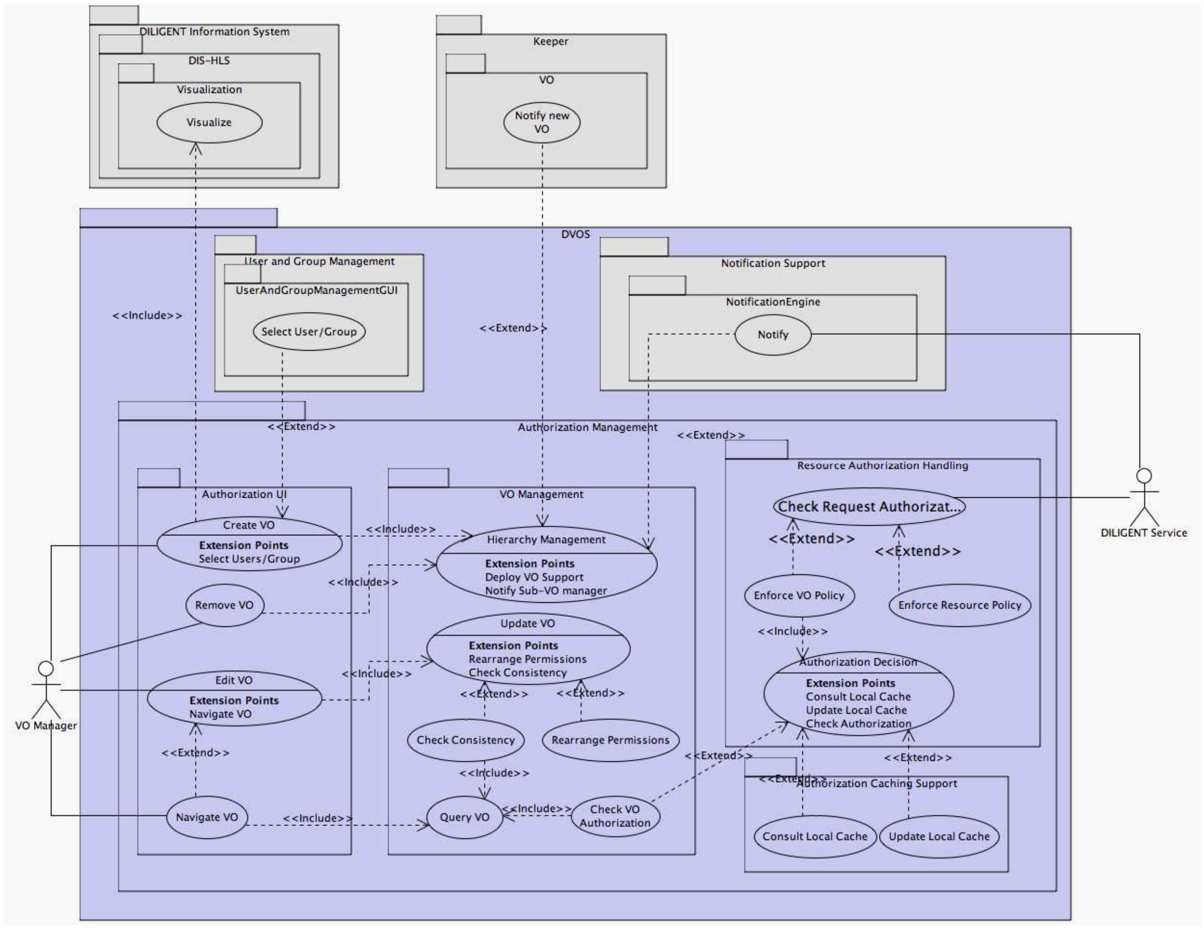


Figure 34. Authorization Management – Use Case View

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**

  It enables VO Managers to create and remove sub-VOs interacting for resource and user selection and inclusion. These use cases 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 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".

VO Management package

This package is the core of the VO Authorization model, it contains entities responsible to model structure and behavior of DILIGENT VO and VO Hierarchy.

- **Check Consistency**

Any time permission is granted to a role over a given resource, the same sharing rules, defined by the resource manager on that resource, need to be respected. This use case models such a consistency check, which guarantees that no role (and thus no user) can use a resource beyond the authorization granted to the whole VO.

- **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 doesn't fulfill any sharing rules is removed.

- **Check VO Authorization**

  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**

  Query VO is high-level inspection functionality and, according to D1.1.1 Functional Specifications [1], it comprises a number of sub-functionalities for the investigation of the VO status.

- **Update VO**

  Update VO is high-level management functionality and, according to D1.1.1 Functional Specifications [1], it comprises a number of sub-functionalities for the modification of the VO status.

  Among these functionalities *SetRolePermissions* needs to be mentioned because it's used to grant resource usage permissions to DILIGENT users. In particular, the *checkConsistency* functionality is used any time a new permission is granted in order to respect sharing rules set by Resource Manager (see 7.5 Resource Registration Support).

- **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 for deployment actions of new services needed by the new VO (see Figure 19).

  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 is preceded by notifications to all the VO Managers of the sub-VOs.

Resource Authorization Handling package

DILIGENT services need a set of functionalities in order to enforce locally the VO-level authorization policies. This package provides support for these functionalities.

- **Check Request Authorization**

  This is the core functionality used by services in order to enforce authorization policies on resources. While DILIGENT services and resources mostly exploit VO-level Authorization model, the enforcement of Resource-level policies as extension point of this functionality allows the adoption of alternative authorization models (e.g. for legacy resources).

- **Enforce VO Policy**

This functionality models the enforcement of the VO-level authorization policies. The outcome of such a decision depends exclusively on the resource usage policies defined within the VO. The enforcement of the VO-level policy relies on the *Authorization Decision* functionality (see below).

- **Enforce Resource Policy**

  Beside VO Policy, Resources can apply local policy to user requests. This is often the case for a resource that applies some fine-grained authorization policies on the resource itself. (e.g., a content manager should control access on individual content objects, which is not managed at VO level).

- **Authorization Decision**

  This functionality return an authorization decision based on the triple <user, resource, action>. This functionality can either consult the local cache or query the VO Management through the *Check VO Authorization* functionality.

Authorization Caching Support package

In a distributed system where authorization information is spread over network nodes the adoption of a caching mechanism can reduce the time needed to enforce VO-level policies and improve performance. The adoption of such a mechanism is not mandatory for DILIGENT services, but is recommended for services whose authorization information changes rarely.

- **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.

- **Update Local Cache**

  Local caches must be updated to avoid inconsistency with authorization information managed by the VO Management package. Each DILIGENT Service can set its own cache update frequency. Expired authorization information is automatically deleted from the cache, but new information for a given resource is reloaded only when a new request for the resource is received.

  Furthermore we are investigating the use of a notification model to forward policy changes from Authorization server directly to local caches.

7.2.2 Logical View

Some entities are involved in providing functionalities described in paragraph 7.2.1:

- A federation of *Authorization Resources* is 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.
- On each DHN a *Cache Manager* is in charge to maintain a local copy of authorization information for DILIGENT Resources hosted by the node. Authorization information is obtained contacting the federation of *Authorization Resources*. *Cache Managers* provide functionalities included in the Authorization Caching Support Package.
- An instance of *VOLevelPDP* handler for each DILIGENT service is in charge of making *Authorization Decisions* contacting the local *Cache Manager*.
- A user Interface provides VO Managers with access to *Authorization Resources*. This interface supplies functionalities of the Authorization UI package.

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

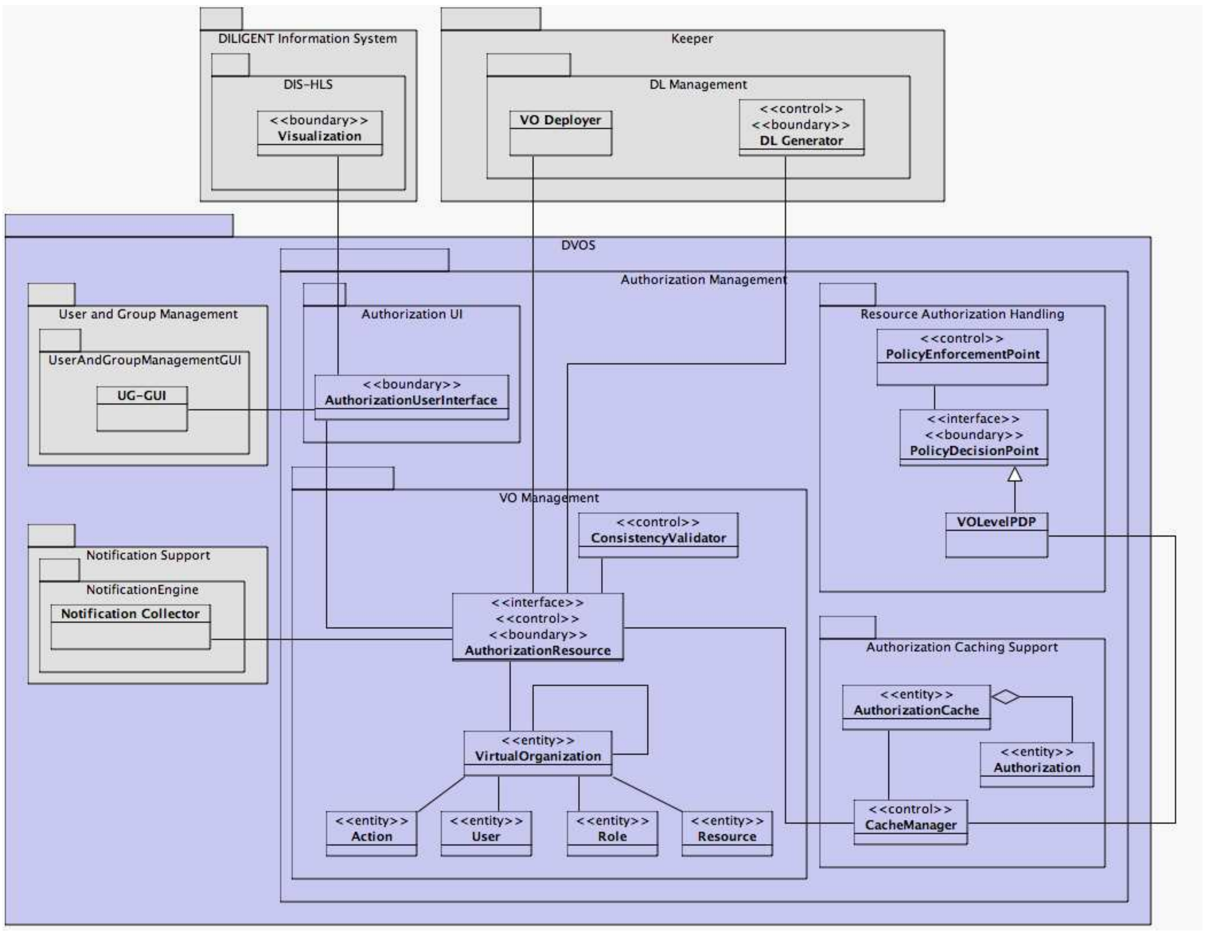

DILIGENT Information System
DIS-HLS
<<boundary>> Visualization
Keeper
DL Management
VO Deployer
<<control>> <<boundary>> DL Generator
DVOS
Authorization Management
User and Group Management
UserAndGroupManagementGUI
UG-GUI
Authorization UI
<<boundary>> AuthorizationUserInterface
Resource Authorization Handling
<<control>> PolicyEnforcementPoint
<<interface>> <<boundary>> PolicyDecisionPoint
VOLevelPDP
VO Management
<<control>> ConsistencyValidator
Notification Support
NotificationEngine
Notification Collector
<<interface>> <<control>> <<boundary>> AuthorizationResource
<<entity>> VirtualOrganization
<<entity>> Action
<<entity>> User
<<entity>> Role
<<entity>> Resource
Authorization Caching Support
<<entity>> AuthorizationCache
<<entity>> Authorization
<<control>> CacheManager

Figure 35. Authorization Management – Logical View

A short description of classes in each package is provided below.

AuthorizationUserInterface

This class is a graphical user interface to VO Management logic. It allows users to manually inspect and modify VO and VO Hierarchy. It interacts with DIS visualization package and User Management package to discover and select users and resources to be included/removed in/from the VOs.

ConsistencyValidator

It provides consistency between Permissions and Sharing Rules in each VO.

[to be detailed in D1.2.2]

AuthorizationResource

AuthorizationResource are in charge of maintaining and providing VO-level authorization information (i.e. Users, Roles, Permissions and Sharing Rules). Each *AuthorizationResource* represents a single VO in the DILIGENT VO hierarchy and, when asked, it supplies a "local to the VO" view of this information. Asking an *AuthorizationResource* for authorization information will return results related to the VO-tree rooted at that VO. E.g. asking for Users belonging to the VO will return all the users of the VO, including all the users of the sub-VOs[12]. In order to assure global consistency in the VO hierarchy *AuthorizationResources* are linked in a tree-like structure rooted at DILIGENT VO.

[12] For sub-VO inheritance see paragraph 3.7 of D1.1.1 "DILIGENT Functional Specifications"

Every time a new VO is created a new *AuthorizationResource*, representing the new VO, must be instantiated. This task is accomplished with the support of the *ServiceDeployer* class provided by the keeper service (see Figure 23).

AuthorizationResource also interacts with *NotificationCollector* during VO removal operation in order to Notify VO Managers of the sub-VO

[to be detailed in D1.2.2]

VirtualOrganization (Action, Role, User, Resource)

These classes represent the VO internal model.

[to be detailed in D1.2.2]

PolicyEnforcementPoint

This class is in charge to manage the authorization chain during handling of incoming requests. The Java WS Core service container provides default implementation of this class. The access to the service/resource is granted to the request if all the PolicyDecisionPoint in the configured authorization chain return a permit.

[to be detailed in D1.2.2]

PolicyDecisionPoint

Interface implemented by service handlers in order to implement authorization decision logics. An authorization chain can be configured for each service/resource in order to return authorization decisions based on service/resource-specific authorization models.

VOLevelPDP

This is the implementation of the PolicyDecisionPoint interface provided by the Authorization Management. This handler is in charge to make authorization decision based on DILIGENT VO-level authorization model. This handler contacts the local *CacheManager* in order to obtain authorization information and return a permit if and only if the requesting user is allowed to perform the requested action on the specified resource.

AuthorizationCache

The AuthorizationCache contains a local (to the DHN) copy of VO-level authorization information. This information is useful to speed up the authorization decisions of the VOLevelPDP.

CacheManager

This class is in charge to supply the VOLevelPDP with authorization information needed to make authorization decision. It is also in charge to manage authorization information contained in the local cache.

7.2.3 Deployment View

The deployment diagram below shows a possible configuration for *AuthorizationResources* and for the corresponding *AuthorizationUserInterfaces*. In this diagram two VO are represented: an *Arte VO* and a sub VO named *DL1*.

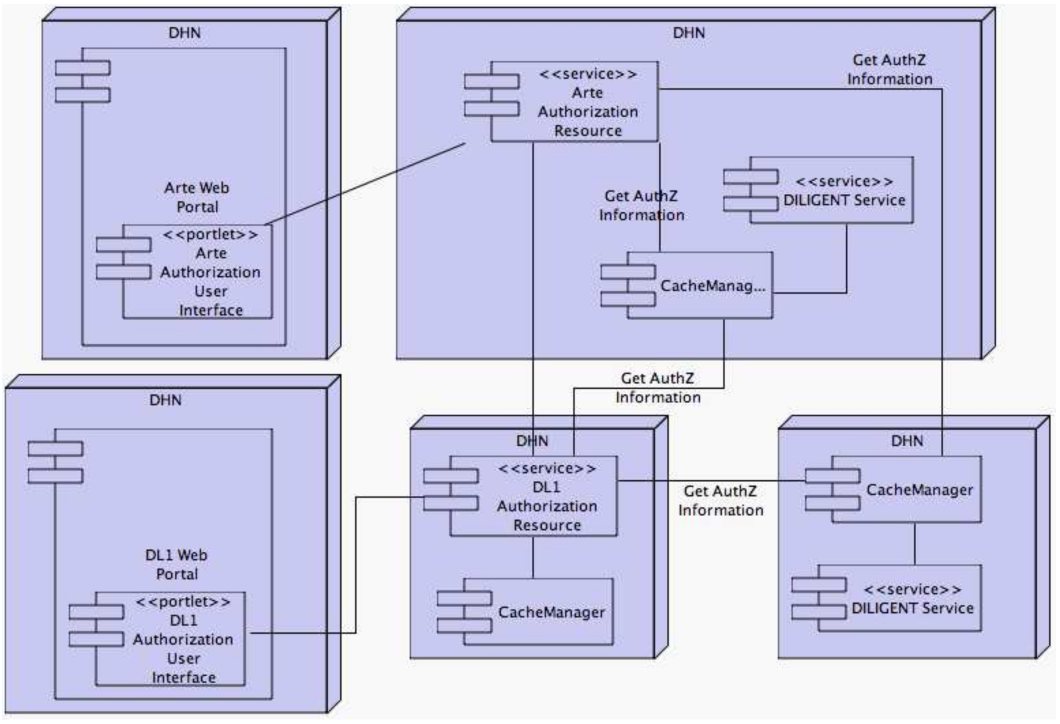

DHN
Arte Web Portal
<<portlet>> Arte Authorization User Interface
DHN
<<service>> Arte Authorization Resource
Get AuthZ Information
Get AuthZ Information
<<service>> DILIGENT Service
CacheManag...
Get AuthZ Information
DHN
DL1 Web Portal
<<portlet>> DL1 Authorization User Interface
DHN
<<service>> DL1 Authorization Resource
CacheManager
Get AuthZ Information
DHN
CacheManager
<<service>> DILIGENT Service

Figure 36. Authorization Management – Deployment View

In order to access and manage a *VO AuthorizationResource* a corresponding *AuthorizationUserInterface* must be deployed in a Web portal. For VOs that model DLs the corresponding portlet can be deployed in the DL Web Portal itself. For others VOs the corresponding *AuthorizationUserInterface* portlets are deployed in the DILIGENT Web Portal.

There is a *Cache Manager* service for each DILIGENT Node; *VOLevelPDP* handlers of DILIGENT Services hosted by the DHN contact the *Cache Manager* of the local node to obtain authorization information. These *VOLevelPDP* handlers are not shown in Figure 36 for clearness reasons.

[to be detailed in D1.2.2]

7.2.4 Significant Classes and components

[to be provided in D1.2.2]

7.3 Notification Service

7.3.1 Use-Case View

In D1.1.1 [1] are identified three functionalities related to notification area:

- Notify a User – enables a generic DILIGENT Service to notify a user by its own identity
- Notify a Group – enables a generic DILIGENT Service to notify all users belonging to a given group
- Notify a Role – enables the notification to all users having a particular Role

The functional specification document identifies a DILIGENT service as the publisher of notification messages.

The typical consumer of a notification is always a human actor.

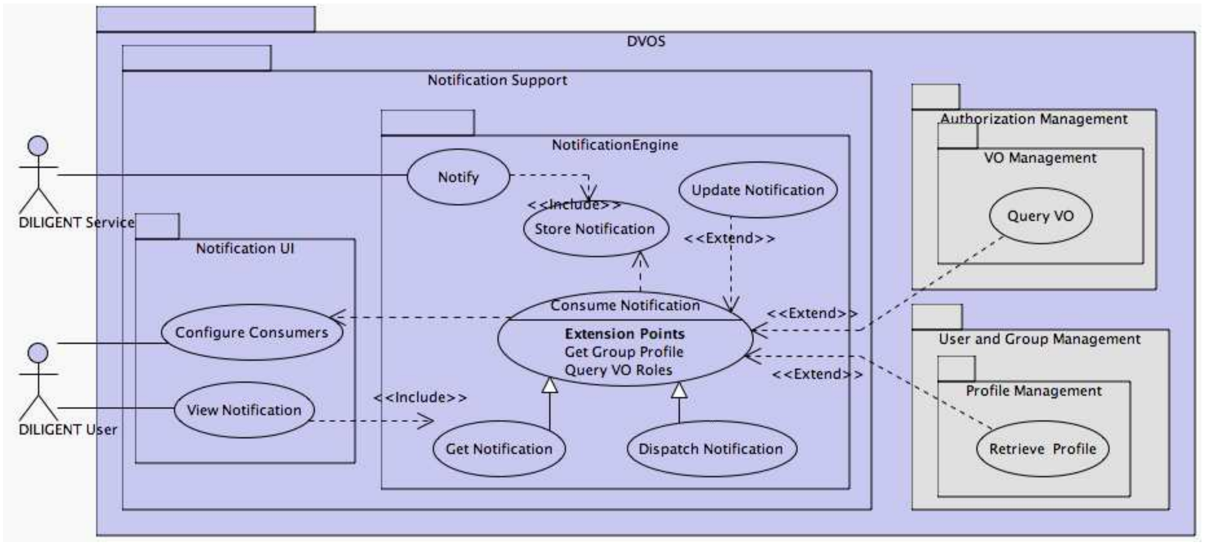


Figure 37. Notification Support – Use Case View

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

NotificationEngine package

- **Notify**

  The functionality enabling an actor to start a notification process. As mentioned above, the target of a notification can be a single user, a group of users or a set of users with a given role.
- **Store Notification**

  After its creation a notification message needs to be stored along with its status before any delivery action can be initiated. This step is peculiar if notification delivery is performed adopting a 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 *UsersManagement* are also exploited.
- **Dispatch Notification**

  It 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)
- **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

- **Configure Consumers**

  It enables a user to change its own delivery preferences (e.g. email, web client).

- **View Notification**

  This functionality either realized through a portlet or through a web client, enables users to directly pull notifications from the storage.

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.

[to be detailed in D1.2.2]

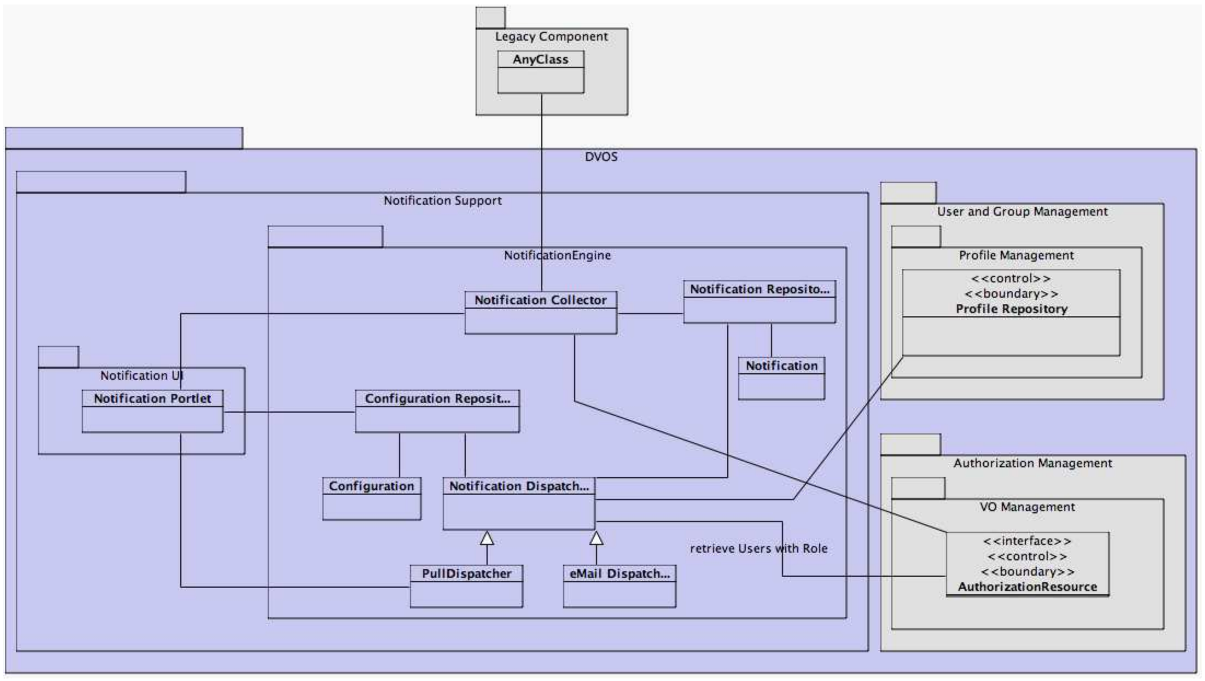


Figure 38. Notification Support – Logical View

Notification Engine package

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).

[to be detailed in D1.2.2]

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.)

[to be detailed in D1.2.2]

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 and is hosted by the DILIGENT Portal Engine.
- Notification Service: a WSRF service hosted on any DILIGENT Hosting Node, which is in charge to implement the Notification Engine package behavior.

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 endpoint reference to the NotificationUI for retrieval and configuration.

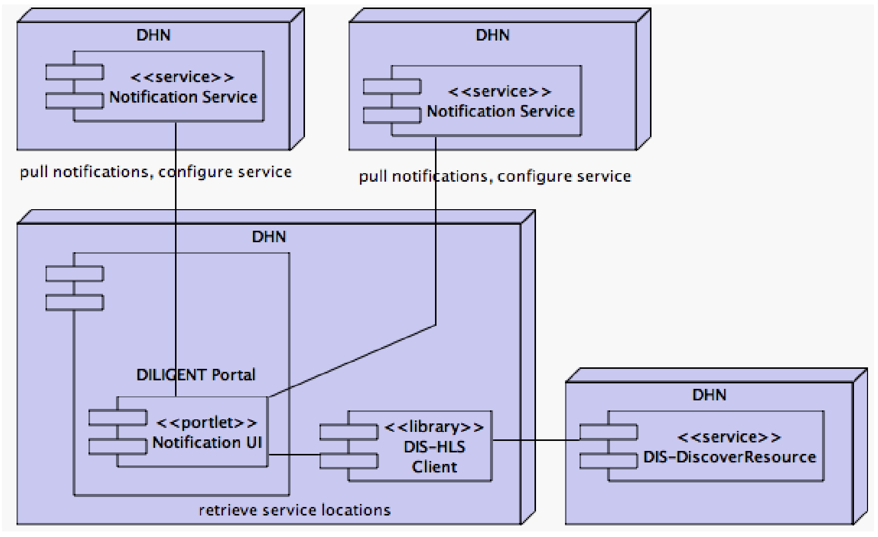


Figure 39. Notification Support - Deployment View

[to be detailed in D1.2.2]

7.3.4 Significant Classes and components

[to be provided in D1.2.2]

7.4 User and Group Management Service

Accordingly 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: *user management*, *group management*, *search*, and *invitation.*

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

The *group management* part includes use cases to create and remove groups of users as well as to edit the group profiles. The model underlying the adjunction of a user to a group implies that a user is invited to join a group; then the invited user accepts or rejects the invitation. Finally, 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 invited 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.

[to be detailed in D1.2.2]

7.4.1 Use-Case View

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

- Users Management
  - 4.5.7 Add a User to DILIGENT

- 4.5.12 Remove a User from DILIGENT

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

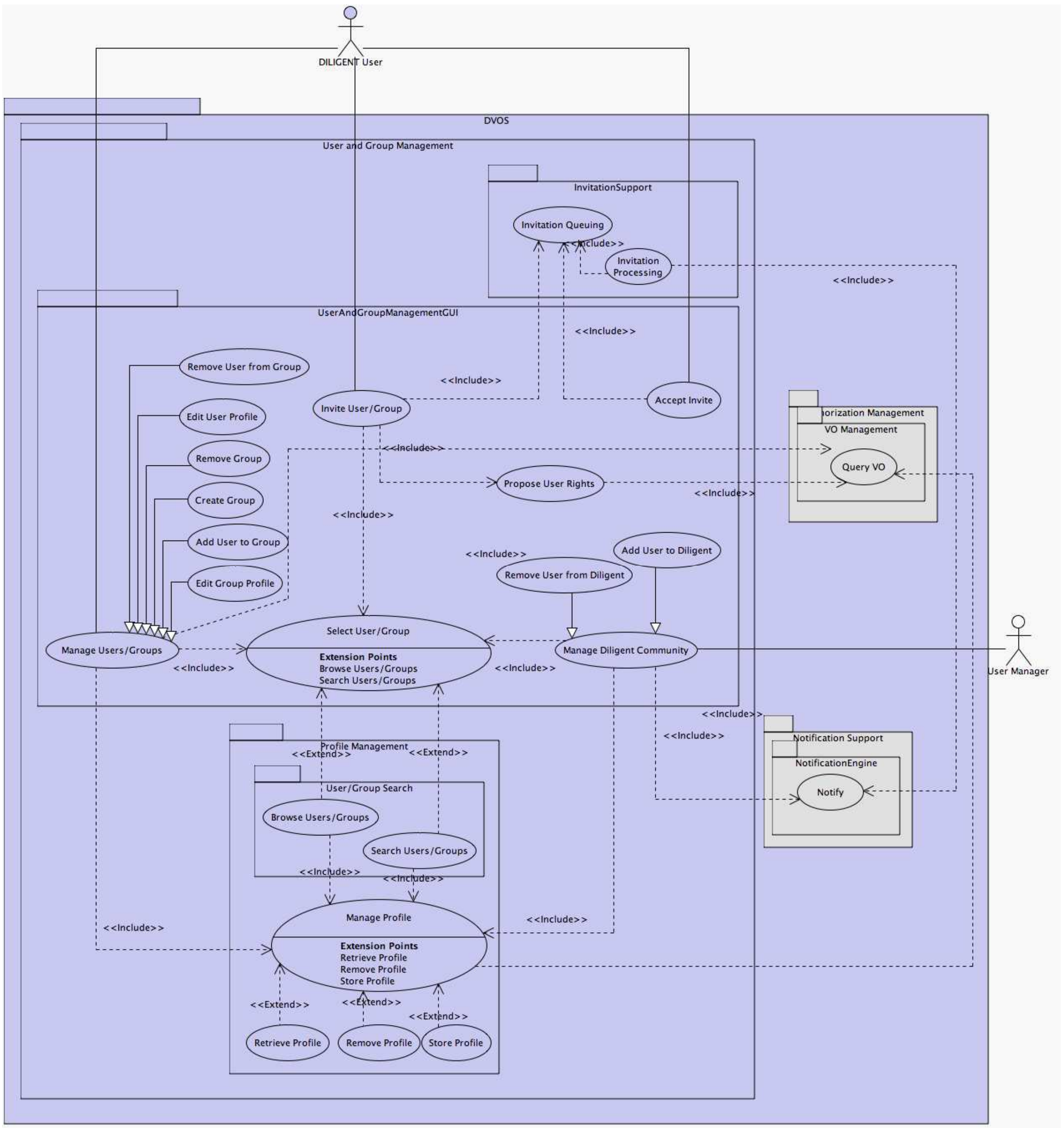


Figure 40. Users Management – Use Case View

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 and discharge of users and groups in the DILIGENT Community. He's 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 his own groups, modify 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 Specifications" [1]. With respect to D1.1.1 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 he's invited to use. Finally, 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 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.

[to be completed in the D1.2.2]

7.4.2 Logical View

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

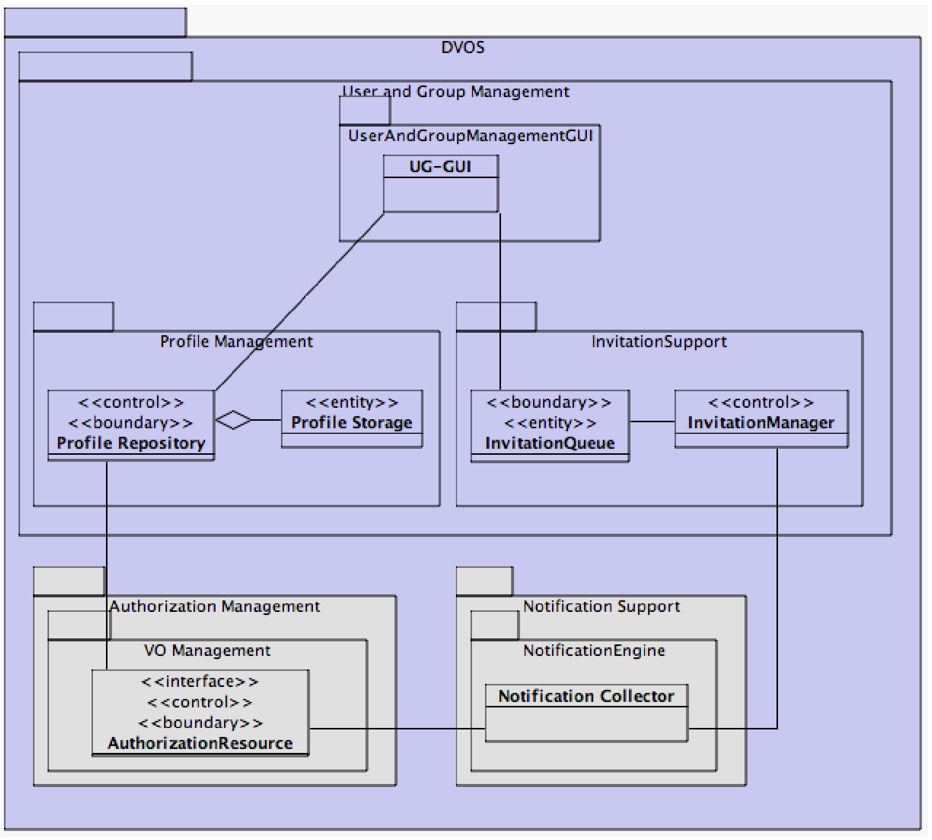

DVOS
User and Group Management
UserAndGroupManagementGUI
UG-GUI
Profile Management
<<control>> <<boundary>> Profile Repository
<<entity>> Profile Storage
InvitationSupport
<<boundary>> <<entity>> InvitationQueue
<<control>> InvitationManager
Authorization Management
VO Management
<<interface>> <<control>> <<boundary>> AuthorizationResource
Notification Support
NotificationEngine
Notification Collector

Figure 41. 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 *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*)

[to be detailed in D1.2.2]

7.4.3 Deployment View

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.

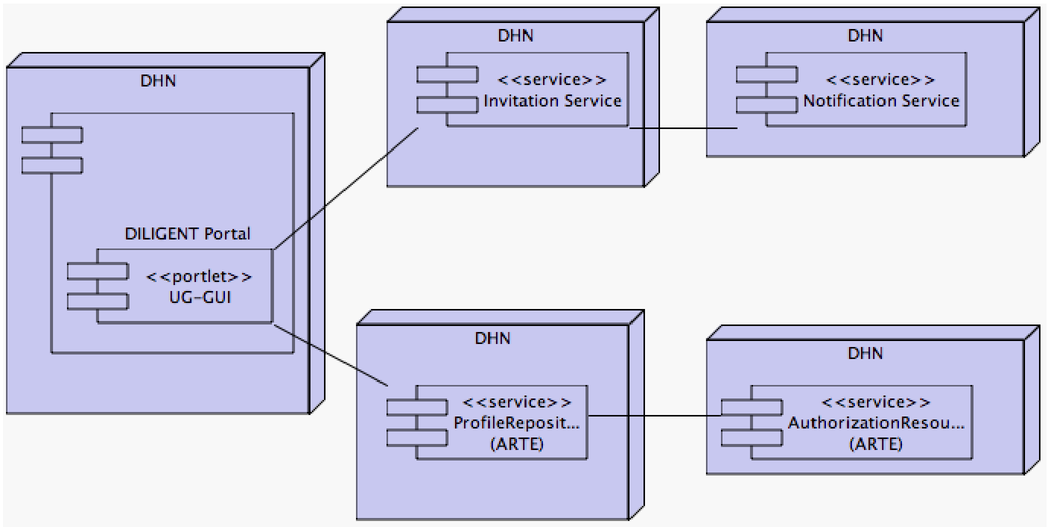


Figure 42. User and Group Management - Deployment View

[to be detailed in D1.2.2]

7.4.4 Significant Classes and components

[to be provided in D1.2.2]

7.5 Resource Registration Support

Resource Registration Support provides mechanisms enabling resource owners to allow the DILIGENT community to use their own resources. The DILIGENT infrastructure imposes several constraints on resource usage, where the most important are the conformity to the DILIGENT Data 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.

7.5.1 Use-Case View

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

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

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

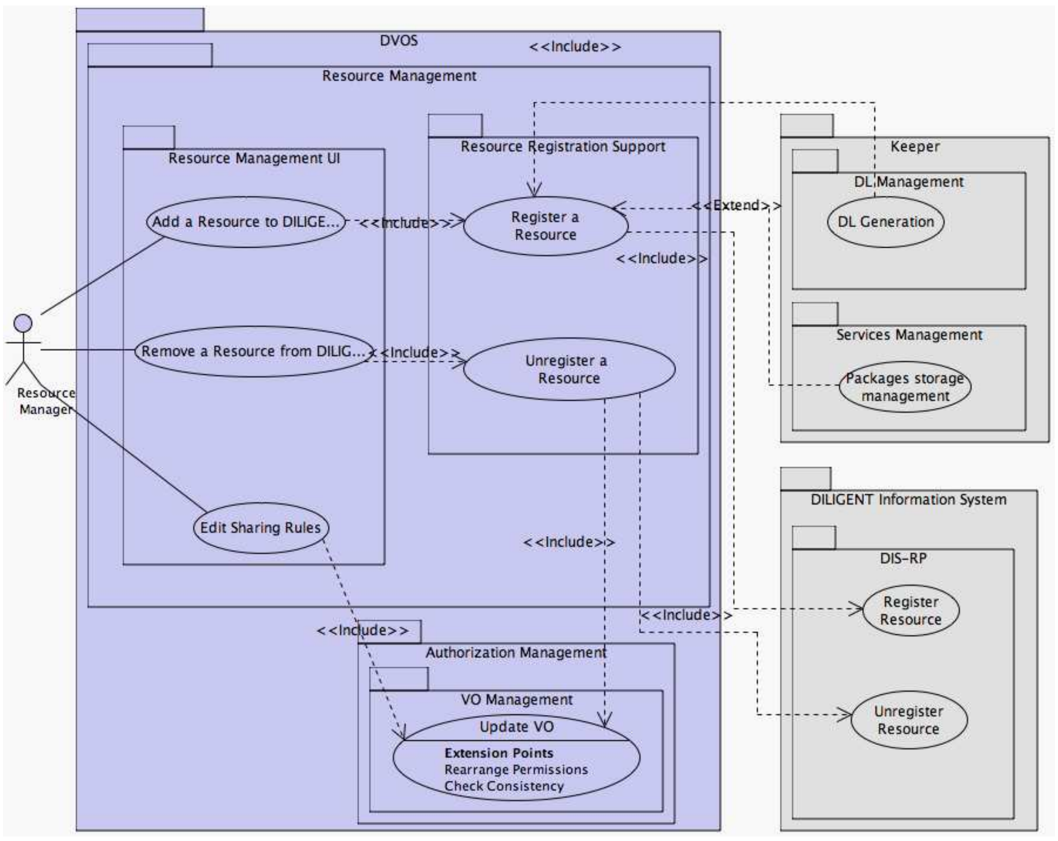


Figure 43. Resource Registration Support – Use Case View

Add a Resource to DILIGENT

This functionality deals with the registration of a new resource within the DILIGENT infrastructure, thus allowing the Resource Manager to describe the resource being registered, deciding which VOs will be enabled to use the resource (the DILIGENT VO is the default) and ruling the use of the resource by imposing a set of sharing rules on it.

Register a Resource

This functionality actually models the registration workflow. This workflow depends on the type of resource being registered in the DILIGENT infrastructure. The differentiation is needed in order to allow manual and automatic registration models as required by different DILIGENT Resource types. E.g. the registration model for a resource of type *Collection* should be performed without VO Manager intervention, rather than the registration of a new *Archive* Resource. A registration workflow, involving VO Manager manual intervention, can be summarized as follow:

- Formal verification of the resource profile. This includes a formal check of the information supplied with regard to the DILIGENT Resource Data Model defined in Section 3. If the resource being registered is a Package, the Keeper Service is also asked to validate the package (see Section 6).
- Registration of the new Resource in the DILIGENT Information Service.
- Notification to various VO managers in order to include the new resource in their VOs.
- Finally, the Resource Manager is in charge of defining the sharing rules for the new resource with respect to the various VOs the resource has been added to.

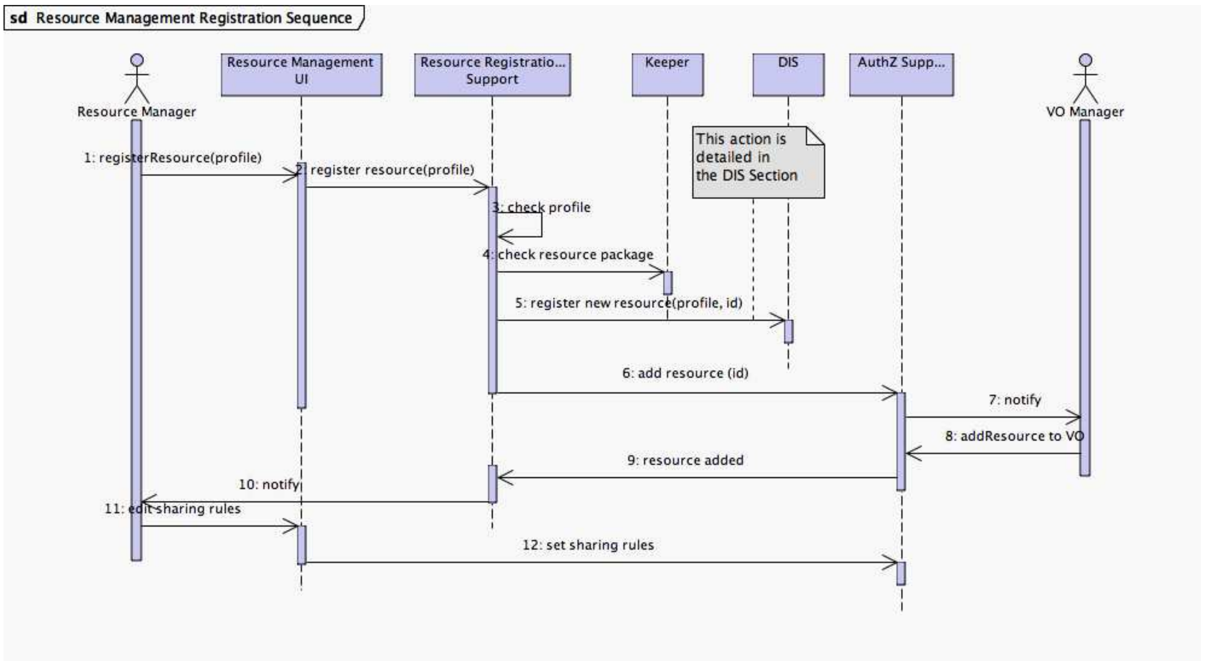

sd Resource Management Registration Sequence
Resource Manager
Resource Management UI
Resource Registratio... Support
Keeper
DIS
AuthZ Supp...
VO Manager
This action is detailed in the DIS Section
1: registerResource(profile)
2: register resource(profile)
3: check profile
4: check resource package
5: register new resource(profile, id)
6: add resource (id)
7: notify
8: addResource to VO
9: resource added
10: notify
11: edit sharing rules
12: set sharing rules

Figure 44. Register a Resource - Sequence Diagram

Remove a Resource from DILIGENT

This functionality definitely unregisters the resource from the DILIGENT infrastructure; as a consequence, the resource is removed from each VOs using it.

Unregister a Resource

Similarly to 'Register a Resource', this functionality models the workflow underlying the removal of a resource from DILIGENT. As well as for the registration, the unregistration workflow also depends on the type of the resource to be removed. An unregistration workflow, involving VO Manager manual intervention, can be summarized as follow:

- Sharing rules are removed each interested VO. Consequently, the Authorization Support Service will rearrange permissions on the resource accordingly, thus revoking any right on it. From this point on, the resource is actually no longer available to users.
- VO Managers of interested VOs are notified in order to remove the resource from their VOs.
- Finally, the resource profile is removed from the DILIGENT Information Service.

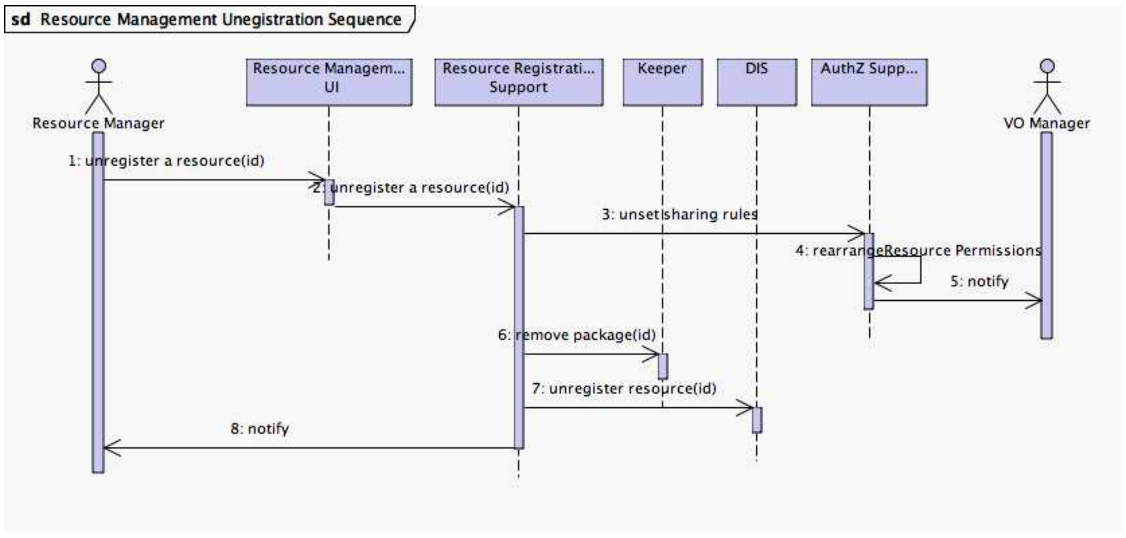

sd Resource Management Unegistration Sequence
Resource Manager
Resource Managem... UI
Resource Registrati... Support
Keeper
DIS
AuthZ Supp...
VO Manager
1: unregister a resource(id)
2: unregister a resource(id)
3: unset sharing rules
4: rearrangeResource Permissions
5: notify
6: remove package(id)
7: unregister resource(id)
8: notify

Figure 45. Unregister a Resource - Sequence Diagram

Edit Resource Sharing Rules

Sharing rules define the way a VO is entitled to use a resource. This functionality enables the resource manager to define them in terms of a whole VO regardless of its members. It can be noticed that definition of sharing rules is part of the registration use case. Anyway, Resource Managers can change them any time during the resource lifetime; this doesn't directly change permissions on the resource, but triggers an action on Authorization Support Service in order to rearrange permissions accordingly (see section 7).

Manage a Resource in a DL

From the point of view of the Dynamic VO Support, the management of a resource within a Digital Library consists of managing the resource within the scope of the underlying Virtual Organization. A description of functionalities related with VO management can be found in section 7.2 Authorization Management.

7.5.2 Logical View

An in-depth analysis of previous Use Case diagram leads to the following Logical View. Figure 46 shows main classes and components involved in resource registration and management. Interactions with others Collective Layer service areas are also indicated. A brief description of each class is provided below.

[to be detailed in D1.2.2]

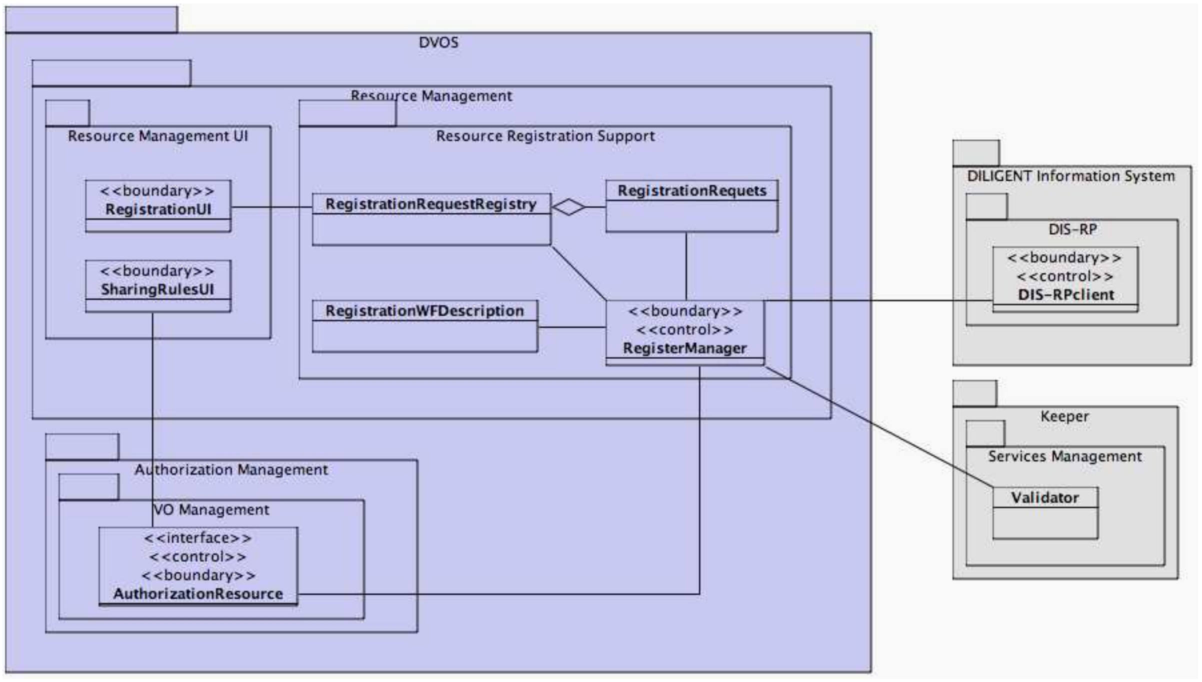

DVOS
Resource Management
Resource Management UI
<<boundary>> RegistrationUI
<<boundary>> SharingRulesUI
Resource Registration Support
RegistrationRequestRegistry
RegistrationRequets
RegistrationWFDescription
<<boundary>> <<control>> RegisterManager
DILIGENT Information System
DIS-RP
<<boundary>> <<control>> DIS-RPclient
Keeper
Services Management
Validator
Authorization Management
VO Management
<<interface>> <<control>> <<boundary>> AuthorizationResource

Figure 46. Resource Registration Support – Logical View

RegistrationUI

Through this user interface Resource Managers are enabled to register (and unregister) resources in (from) the DILIGENT infrastructure. Registration (and unregistration) protocol are described in 7.5.1 Use-Case View

SharingRulesUI

Through this user interface Resource Managers are enabled to modify sharing rules for resources previously registered in DILIGENT infrastructure. Sharing Rules are stored in *AuthorizationResource* hierarchy and can be set on a per VO basis, allowing Resource Managers to grant different authorizations to different VO in the hierarchy.

RegistrationRequestRegistry

Registration and unregistration requests submitted by Resource Managers are sent to *RegistrationRequestRegistry* component, which is in charge to maintain status of incoming requests and to submit new or modified requests to the *RegistrationManager* component.

RegistrationManager

RegistrationManager acts as a consumer of the *RegistrationRequestRegistry* component. It is in charge to elaborate new or modified requests performing steps described in the *RegistrationWFDescription*.

RegistrationRequest

This entity element models single registration (or unregistration) request and its current status. The status of requests maintained in the registry is updated by the *RegistrationManager* component based on actions performed.

RegistrationWFDescription

This entity element contains a formal description of registration and unregistration processes. *RegistrationManager* component exploits this information to perform resource registration and unregistration. Separation between description and execution of registration process is needed to exploit different registration processes depending on the types of resources to be registered in DILIGENT infrastructure (e.g. registration of a new *Collection* is a more “light” and automatic process than registration of a new DILIGENT Hosting Node).

[to be detailed in D1.2.2]

7.5.3 Deployment View

Figure 47 shows a deployment view of the Resource Management component.
[to be detailed in D1.2.2]

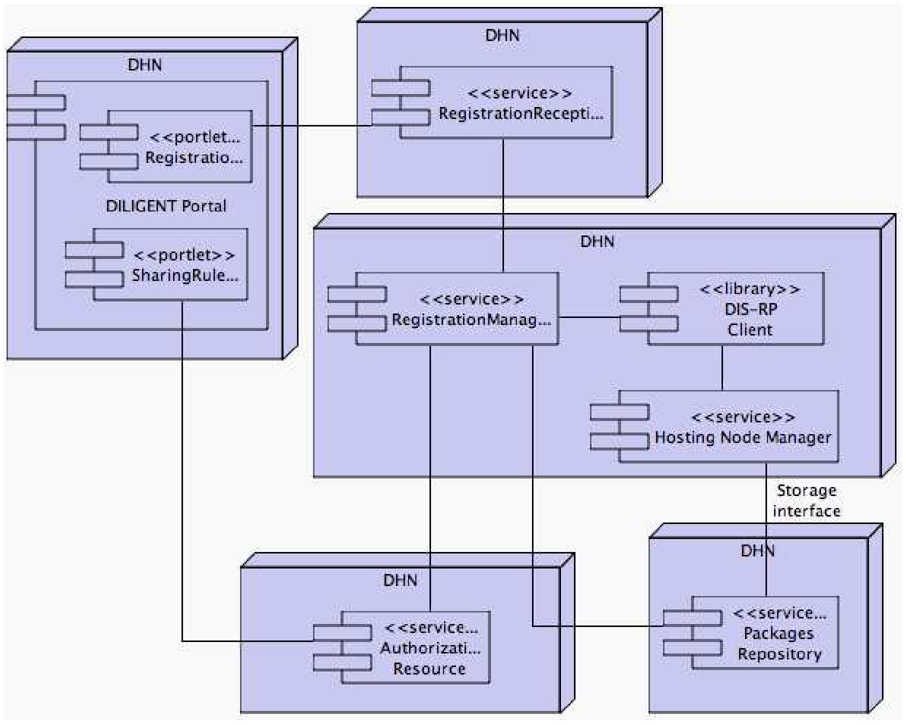


Figure 47. Resource Registration Support – Deployment View

The *RegistrationManager* component exploits local DIS-RP Client API support (see Section 4) in order to notify the Information Service of the existence of the new DILIGENT Resource.
[to be detailed in D1.2.2]

7.5.4 Significant Classes and components

[to be provided in D1.2.2]

8 VDL GENERATOR SERVICE

The VDL Generator Service is the service that enables users/communities to create their own DLs. It receives a set of criteria that specify the expected characteristics of the new DL and 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 functionality, the QoS;
- Selects the more appropriate pool of services and information sources required to implement a DL that satisfies the specified criteria;
- Notifies to the Keeper service the identified services.[13]

This service is thus characterized by a high interaction with a user by a graphical user interface. When specifying the architecture of such kind of interactive service, the challenge is to keep the functional core independent of 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 is our base design choice and 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.

8.1 Use-Case View

Analyzing the deliverable D1.1.1 "Test-bed Functional Specification" the following system functionalities have been identified as related with the VDL Generator Service. In particular, the following functionalities, described as UCs in D1.1.1, will be implemented[14]:

- 4.2.1 Define a DL
- 4.2 Select Resources
- 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

The following functionalities represent those used by the VDL Generator Service to support its tasks:

- 4.3.18 Browse Available Resources
- 6.6.2 Search Archive
- 6.6.2 Search for Object/Resource
- 4.5.26 Invite a Group to a DL

[13] From the DoW.
[14] In bracket we report the D1.1.1 Paragraph number where the UCs is described.

- 4.5.21 Invite a User to a DL
- 4.6.2 Notify Role

These functionalities are rearranged into packages and use cases in Figure 48. The same diagram also reports the dependencies and relationships with other packages. In our re-organization: i) use cases having the same name of those reported in the list represents the same functionality, ii) the functionalities related with 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 graphical user interface functionality (VDLGeneratorUI) and the VDLGeneratorLogic package, iii) other UCs are new, i.e. they do not come directly from any user functionality but are needed to support them, e.g. Query KB. Each UC is described in the following subsections.

VDLGeneratorUI

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

- **Define a DL**

  This use case represents the activity the DL Designer performs by the graphical user interface in order to specify the characteristics of the DL. This activity is performed by the graphical user interface that, with the VDLGeneratorLogic support, is capable to prevent the DL Designer to make inconsistent choices.

  This functionality also allows to specify users that are entitled to have access to DL by using the Select User/Group functionality offered by User and Group Management in Section 7.4.
- **Select Resources**

  This use case, interacting with the DLGenerator, is in charge to propose, step-by-step, the pool of resources that can be used to fulfil the user requirements. For instance, by this UC the DL Designer is enabled to select the archives the 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 DL definition choices performed.
- **Modify a DL**

  This use case represents the activity the DL Designer performs by the graphical user interface in order to update/review the characteristics of a previously defined.
- **Dispose a DL**

  This use case represents the activity the DL Designer performs by the graphical user interface in order to remove a previously created DL. 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 use case represents the activity enabling the Dl Designer to establish the actions to perform in order to preserve the status of DL Resources before to remove them. The kind of actions allowed for each kind of resources will be carefully established.

  [to be detailed in D1.2.2]

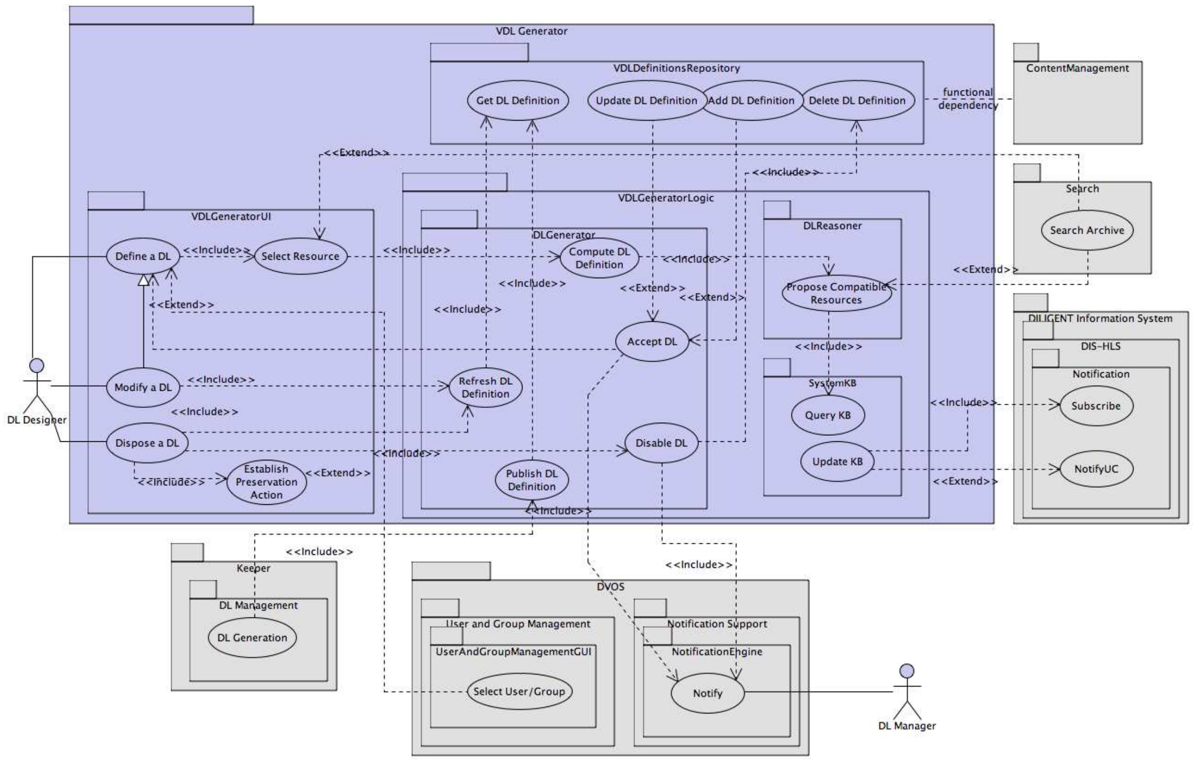


Figure 48. VDL Generator - Use Case View

DLGenerator

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

- **Compute DL Definition**

  This use case represents one of the main activities carried out by the VDL Generator Service, i.e. the computation of the DL Definition in accordance with the selection choices performed by the DL Designer by means of the Define a DL UC. This use case make use of the Propose Compatible Resources described later in order to semi-automatically derive the pool of resources that must be included into the DL Definition.

- **Accept DL**

  This use case represents the act the DL Designer performs in order to make the definition of the DL persistent and to begin the procedure to automatically create the DL. The first action is performed by adding (updating, in case the use case is invoked in the context of Modify a DL UC) the new DL Definition to the definitions repository. The second part of this use case consists in notifying the DL Manager about the request performed by the DL Designer to create (update) the DL.

- **Refresh DL Definition**

  This use case represents the activity performed by the VDL Generator Service in order to recall in line the DL Definition. In fact, as will be 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 use case will thus be invoked each time a previously defined DL must be modified or removed by the DL Designer in order to allow the VDLGeneratorUI functionality to render the DL Definition as stored within the definitions repository.

- **Disable DL**

This use case represents the activity performed by the VDL Generator in order to remove one of the DLs previously defined. As a consequence it is composed by two sub tasks, the invocation of the UC entitled to remove the DL Definition from the definitions repository and the notification to the DL manager about the request, performed by the DL Designer, to remove the DL.

- **Publish DL Definition**

  This use case represents the way by which other services (i.e. the Keeper) can access the computed list of resources forming the DL. This list contains the resources identified as well as their configuration parameters (in case of resources that must be deployed). Moreover the list contains the identifier of users to be enabled to have access to the DL (in the case of restricted access).

DLReasoner

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

- **Propose Compatible Resources**

  This use case represents the algorithm that is in charge to automatically identify the pool of resources to form a complex object as a DL is. This algorithm, by using the knowledge stored within a knowledge base (see the SystemKB), is capable to select and assemble the pool of available components in a suitable way to fulfil the DL requirements. The final product (i.e. the DL) is complicated to obtain because there are potentially thousand of choices and constraint, thus a well-suited solution is needed. From a functional point of view, this functionality is invoked each time a new DL 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 use cases representing the population and the querying of the knowledge base maintained by the VDL Generator about the resources available.

- **Query KB**

  This use case 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.

  [to be detailed in D1.2.2]

- **Update KB**

  This use case represents the 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 UC. The tool used to maintain the knowledge base and the format used to maintain knowledge within the service is hidden to the other DILIGENT services, however it will be expressive enough to represent the kind of information reported in Section 3.

  [to be detailed in D1.2.2]

VDLDefinitionsRepository

This package contains the use cases needed to model the operation for storing, retrieving and removing the DL Definitions.

- **Get DL Definition**

  This use case represents the retrieval of a previously stored DL Definition from the definitions database. Each definition must be complete, i.e. it contains in a separable way the definition as expressed by the DL Designer and the definition as computed by the VDL Generator service.

  [to be detailed in D1.2.2]

- **Add DL Definition**

  This use case represents the adjunction of a new DL Definition to the definitions database. The definition will be stored as files with the aim to use the distributed storage capability offered by the Grid infrastructure. As a consequence the definitions database must be capable to maintain a mapping between the DL identifier and the file/files containing its definition and stored elsewhere.

  [to be detailed in D1.2.2]

- **Update DL Definition**

  This use case represents the updating of a previously stored DL Definition to the definitions database. As previously stated, DL Definitions are stored as files distributed on the storage elements available on the infrastructure. The definitions database maintains the mapping between the DL identifier and the file/files containing the relative definitions, thus this operation must be completed updating all the files accordingly.

  [to be detailed in D1.2.2]

- **Delete DL Definition**

  This use case represents the removal of a previously stored DL Definition to the definitions database. As previously stated, DL Definitions are stored as files distributed on the storage elements available on the infrastructure. The definitions database maintains the mapping between the DL identifier and the file/files containing the relative definitions, thus this operation must be completed removing all the files constituting the DL Definition that are distributed elsewhere.

  [to be detailed in D1.2.2]

8.2 Logical View

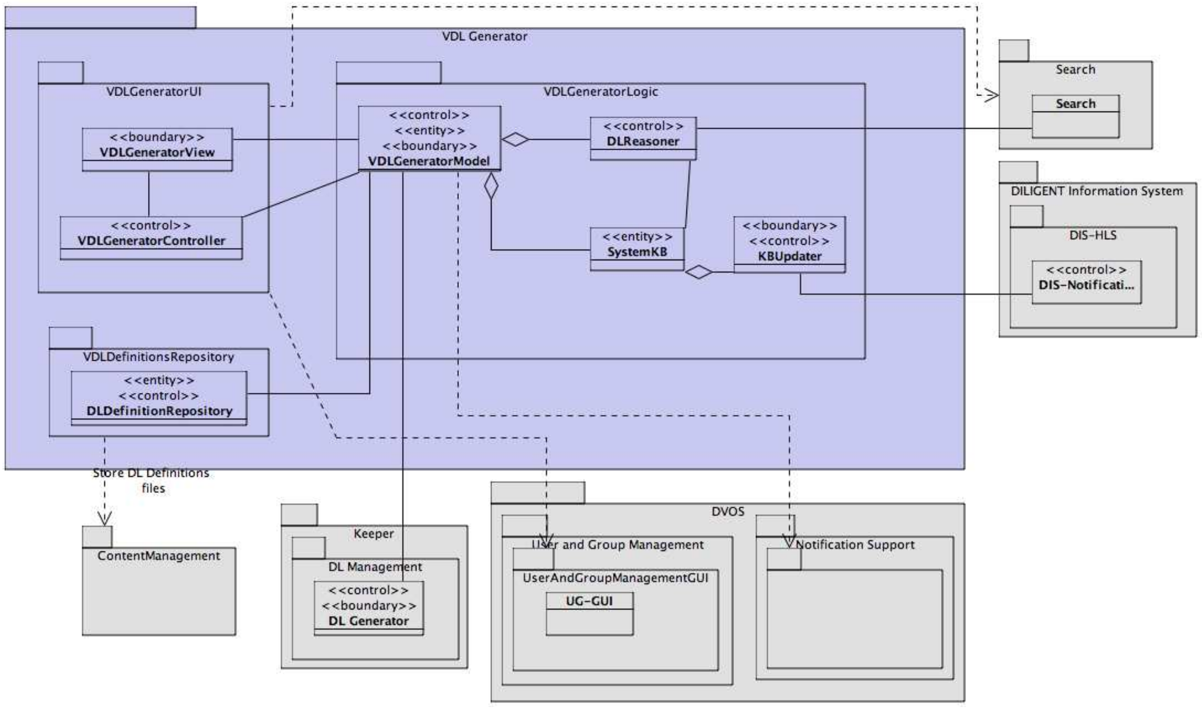
In order to design the VDL generator Service we decided to organize its components following the *Model-View-Controller* architectural pattern [2]. This pattern divides an interactive application into three main components:

- 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.

As we decided to maintain the definitions of all the DLs, DILIGENT is responsible for, we add the *VDLDefinitionRepository*. All these components and the related relationships are shown in Figure 49 and will be described briefly in the following subsection. However, the 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. The VDLGeneratorUI is the package in charge to offer the functionality reported in the use case view by classes reported here as well as the VDLDefinitionsRepository will implement the four functionality using the DLDefinitionRepository class;

- The VDLGeneratorLogic package, organized in sub packages into the use case view, is structured in classes into the logical view; each logical view class represents the use case view package having the same name. Each of these classes is in charge to implement the relative use cases.



VDL Generator
VDLGeneratorUI
<<boundary>> VDLGeneratorView
<<control>> VDLGeneratorController
VDLGeneratorLogic
<<control>> <<entity>> <<boundary>> VDLGeneratorModel
<<control>> DLReasoner
<<entity>> SystemKB
<<boundary>> <<control>> KBUpdater
VDLDefinitionsRepository
<<entity>> <<control>> DLDefinitionRepository
Store DL Definitions files
ContentManagement
Keeper
DL Management
<<control>> <<boundary>> DL Generator
DVOS
User and Group Management
UserAndGroupManagementGUI
UG-GUI
Notification Support
Search
Search
DILIGENT Information System
DIS-HLS
<<control>> DIS-Notificati...

Figure 49. VDL Generator Logical View

8.2.1 VDLGeneratorUI package

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

Details about technologies (e.g. portlets and the usage of WSRP) will be considered as soon as more information and details about the DILIGENT Portal Generator will be available.

[to be detailed in D1.2.2]

8.2.2 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 *DLReasoner*, that represents the components in charge to automatically identify the DL components/resources needed to fulfil the user requirements;

- 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;
- The *AuthorizationController*, that is in charge to implement the interface needed to apply local authorization policy to user request in order to moderate the service access.

Details about the DLReasoner as well as the SystemKB will be supplied in Section 8.4. In order to give a general view, the problem that these classes plan to solve is related with the automatic identification of the most suitable DLComponents needed to fulfil the DL Definition. The solution we adopt is based on the usage of Description Logics [11] to model both the DL Definition criteria and the DLComponents with the related constraint on applicability/usability. By adopting an inference mechanism we are able to identify the set of DLComponents needed to satisfy the DL Definition.

8.2.3 VDLDefinitionsRepository

This package represents the repository in charge to maintain the definitions of the various DILIGENT DLs. These definitions will be stored as XML files. The repository must be capable to capture the DL definition as originally expressed by the user and its rearrangement performed by the DLReasoner in order to express the DL in terms of its forming components. This repository is needed for two reasons:

- Support the "Modify a DL" use case allowing the VDLGeneratorUI to show to the DL Designer the choices performed in the previous definition phase and thus to rearrange them;
- Support 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.

8.3 Deployment View

The deployment of the VDL Generator Service, as shown in Figure 50, is based on a three-layered configuration:

- The *front-end* is mainly composed by the components forming 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 DLReasoner. For reason of performance the latter two components must be maintained on the same node as well as we decided to deploy them on the same node where the VDLGeneratorModel service is deployed;
- The *back-end* is the database in charge to maintain the DL definitions.

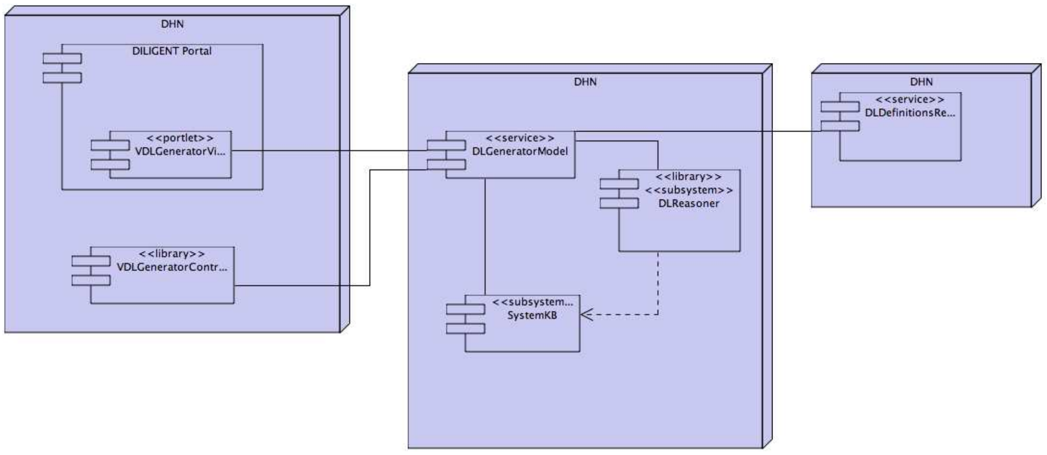


Figure 50. VDL Generator Deployment View

8.4 Significant Classes and Components

8.4.1 VDLGeneratorModel class

As previously stated this class represents the access point to the VDLGeneratorLogic package. As a consequence it is the part of the service in charge to offer the core functionalities of the VDL Generator Service. It is implemented as a Web Service and we assume that an instance of this service is always available within the DILIGENT infrastructure.

To understand how this service will implement the main functionality of the VDL Generator, i.e. the definition of a Digital Library, we present in Figure 51 the sequence diagram showing the various classes involved in and their interactions.

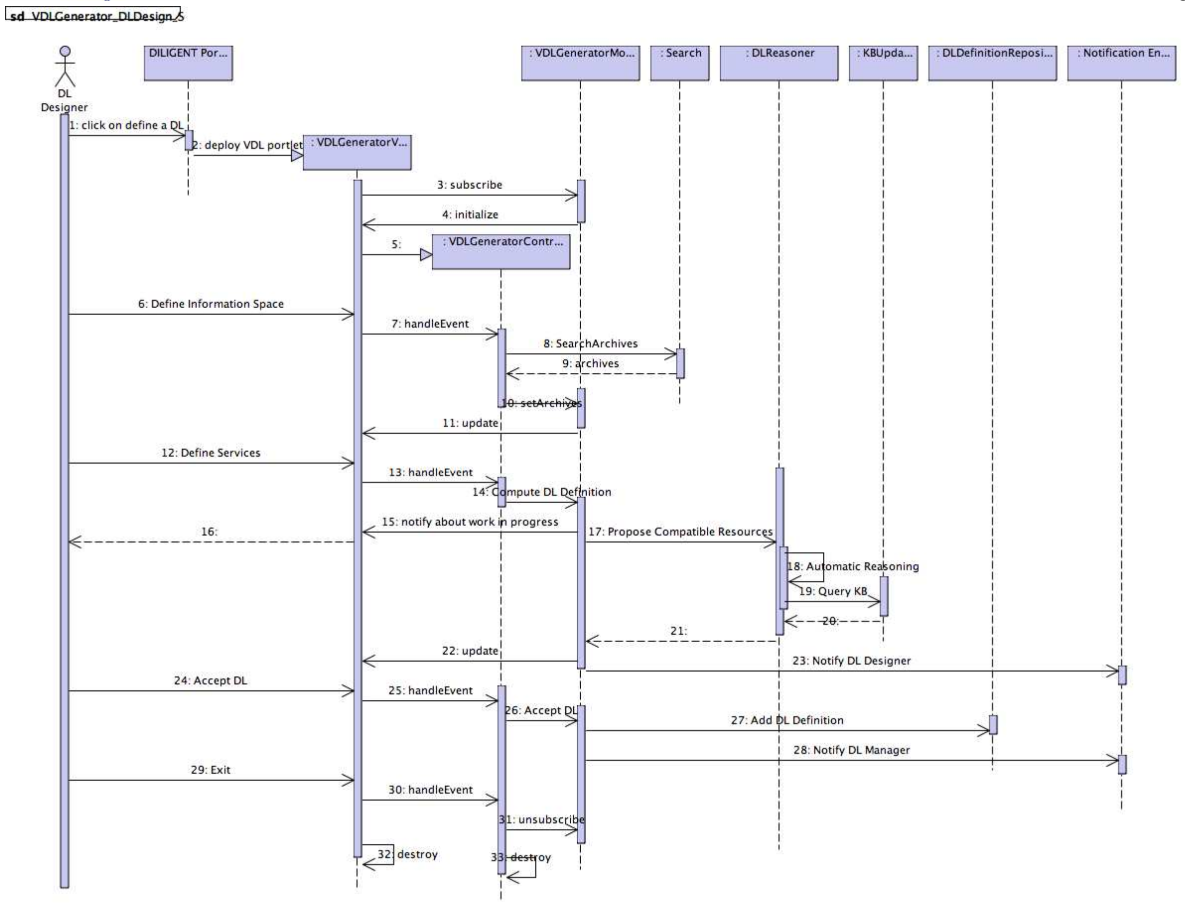


Figure 51. Define a DL

The DL definition process will be organized into the following steps:

1. The DL Designer, after having logged in into the system, is entitled to perform the "create a DL" by clicking on the appropriate link offered by the DILIGENT Portal;
2. The Portal will retrieve the VDLGeneratorView portlet and deploy it in the appropriate way. The deployment mechanism will be detailed by the DILIGENT Portal Engine;
3. The deployment of the portlet is performed by invoking its init method that send a subscribe requests 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 DL. 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. The user interface will be built in a dynamic way;
5. The VDLGeneratorView creates its controller instance that 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 DL he/she is requiring for. The user interface supply two mechanisms to do it:
   - o The DL Designer browse the list of available archives and selects those the DL Community is interested in;

- The DL Designer expresses a characterization criterion that can be passed to the DILIGENT Search service[15] (this service will be 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 expresses strongly depend by the available Search service instance and by its query language. However, the VDLGeneratorModel will supply this configuration aspect to the portlet. For instance, the ARTE community expresses the requirement 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 8 and 9 steps are performed otherwise the VDLGeneratorController skip 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 update its internal partial representation of the DL and the notifies the VDLGeneratorView portlet to update the user interface to take 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 DL he/she is requiring for. Once again the user interface will supply an easy mechanism to express the characteristics of the DL. This strongly depends by the Ontology that we adopt to characterize a DL. Further details are needed. For the time being and in our best knowledge exists only one formal digital library ontology [14]. We plan to define the DILIGENT DL ontology to model DL Definition and DL Components.
13. Once the DL Designer ends the definition phase the VDLGeneratorController is called in order to handle appropriately the event;
14. The VDLGeneratorController calls the VDLGeneratorModel in order to validate, complete and verify the DL definition;
15. In order to avoid the DL Designer to wait in 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 "log out" from the portlet. At the end of the reasoning phase, i.e. when the DL Definition is computed by the system, the DL Designer will be notified in order to enabling he/she to accept/reject the proposed DL;
17. Without human intervention, the VDLGeneratorModel with the support of its DLReasoner and its SystemKB finalizes the definition criteria;
18. Once the DLReasoner is invoked it starts an iterative reasoning process (presented in detail in the next section) whose goal is to verify the correctness of the criteria of

[15] We assume here that 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.

the DL as well as to complete them by identifying all the components needed to build a DL compliant with the specified criteria;

19. To perform its reasoning algorithm the DLReasoner needs to query the SystemKB that will supply information of the available components, mainly their dependencies;
20. The SystemKB will reply to the query following the semantic model underlying the entire decision process. This model is reported in Section 8.4.2.2;
21. At the end of the reasoning process the DLReasoner notifies the VDLGeneratorModel with the definition of the DL obtained;
22. The VDLGeneratorModel updates its internal representation of the DL and the 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 click 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;
29. The user click 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 destroyed;
31. The VDLGeneratorController send an unsubscribe request to the VDLGeneratorModel in order to be removed from the list of subscribers;
32. The VDLGeneratorView portlet is removed;
33. The VDLGeneratorController is removed.

[to be detailed in D1.2.2]

8.4.2 The SystemKB and the DLReasoner classes

8.4.2.1 An introduction to Ontologies and Description Logics

The SystemKB and the DLReasoner are based on Description Logics. Description Logics are formalism for representing knowledge as well as for allowing automatic reasoning about that knowledge [11]. There are two applications of Description Logics that we are interested in:

- They can be used as a conceptual modelling tool for developing ontologies for some universe of discourse (DLs in our case);
- They can be used to build configuration applications, i.e. applications capable to build complex objects (DLs in our case) by assembling components. The VDL generator is a configurator of DLs;

Although the concept of ontology has been around for a very long time in philosophy, in recent years it has become identified with computers as a machine-readable vocabulary that is specified with enough precision to allow differing terms to be precisely related.

More precisely, from the OWL Requirements Document [9]:

An ontology defines the terms used to describe and represent an area of knowledge. Ontologies are used by people, databases, and applications that need to share domain information (a domain is just a specific subject area or area of knowledge, like medicine, tool manufacturing, real estate, automobile repair, financial management, etc.). Ontologies include computer-usable definitions of basic concepts in the domain and the relationships among them [...]. They encode knowledge in a domain and also knowledge that spans domains. In this way, they make that knowledge reusable.

OWL is a Web Ontology Language, i.e. a language for representing ontologies that is compatible with the Web and the Semantic Web. It uses the linking provided by RDF to add the following capabilities to ontologies:

- Ability to be distributed across many systems;
- Scalable to Web needs;
- Compatible with Web standards for accessibility and internationalization;
- Open and extensible.

However, OWL has some expressive limitations that can be onerous in certain application domains, for instance in web services composition where it is necessary to relate inputs and outputs of their component processes [10]. To overcome these limitation languages combining rules and ontologies have been proposed; SWRL is an example [12].

The VDL Generator will make use of these emerging mechanisms as will be explained in the following sections.

8.4.2.2 The Reasoning

The VDL Generator Service, and in particular the functionalities related with the definition of the DL and the automatic identification of the pool of resources needed to fulfil the DL requirements, can be easily modelled as a configuration problem solvable by description logics [11]. 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 known.

All the classical systems for product configuration converge on describing this problem representing two kinds of knowledge: i) *domain description*, i.e. a description of all the types of components available, and ii) the *specification of the desired product*. We will adopt the same approach.

In particular, for domain modeling we will 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 and defined producing an ontology that will be used to annotate all the DILIGENT DL Components. VDL Generator designers will define this ontology as part of their design activity, presented in the *D1.2.2 DL Creation & Management service specification report* and then made available to the other DILIGENT partners enabling them to annotate their services appropriately.

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

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 DLReasoner, 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 DLReasoner 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 DLReasoner 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.

[to be detailed in D1.2.2]

9 CONCLUSION

This report presents the result of the activity conducted within the first period of 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 - May 31st 2005.

The *DL Creation & Management Services Specification interim 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 allows to report main information about the internal behaviour, such as algorithms and detailed information about the data manipulated, use case realization, presented using sequence diagrams, and subsystem.

Finally it is important to remark that this service specification is strongly related 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.

References

[1] DILIGENT. Deliverable No D1.1.1: "Test-bed Functional Specifications"
[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* v1.0 (rev. 14) April 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