Diligent

A DIgital Library Infrastructure on Grid ENabled Technology

Deliverable No D1.1.2:

"Architectural Specification"

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:	D1.1.2
Deliverable title:	Architectural Specification
Contractual Date of Delivery:	Month 15
Actual Date of Delivery:	31st March 2006
Editor(s):	CNR – ISTI
Author(s):	Pedro Andrade, Leonardo Candela, Pasquale Pagano, Manuele Simi
Reviewer(s):	Andrea Manieri (ENG)
Participant(s):	CNR – ISTI
Workpackage:	WP1.1
Workpackage title:	Test-bed functional and architectural design
Workpackage leader:	CNR - ISTI
Workpackage participants:	CNR – ISTI, UoA, UNIBAS, FhG/IPSI, UMIT, CERN, ENG, USG, FAST, 4D Soft Ltd.
Est. Person-months:	28
Distribution:	Confidential
Nature:	Report
Version/Revision:	1.0
Draft/Final	Final
Total number of pages: (including cover)	109
File name:	D1.1.2 Architectural Specification.doc
Key words:	*Digital libraries, UML, System Architecture, System Design*

Disclaimer

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

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

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

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

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

DILIGENT is a project partially funded by the European Union

Table of Contents

Table of Figures

Summary

The aim of this deliverable is to create a common and global vision of the DILIGENT system integrating the service specification reports produced by the WP1.2-1.6. The document focuses on the communications among the different components designed by the technological integration WPs (i.e. WP1.2, WP1.3, WP1.4, WP1.5, and WP1.6) designer teams.

In this direction, it makes a step forward with respect to the release of the D1.1.1 Functional Specification [1], whose goal was to serve as input for the service specification reports. This deliverable concludes the activity of the *WP1.1 Test-bed functional and architectural design*.

Executive Summary

The objective of *D1.1.2 Architectural Specification* is to report the software architecture specification of the whole DILIGENT system. The meaning of "software architecture" can intuitively be understood, however it is difficult to define what a software architecture is. In particular, it is difficult to draw a boundary line between design and architecture, being architecture that part of the design which concentrates on organizational aspects.

In their *An Introduction to Software Architecture* [2], Garland and Shaw suggest that software architecture can be considered as a level of design concerned with the overall system specification:

"*Beyond the algorithm and data structures of the computation; designing and specifying the overall system architecture emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocol for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives*".

However, architecture is more than just structure, e.g. the IEEE Working Group on Architecture defines it as "the highest-level concept of a system in its environment".

The Unified Process methodology, i.e. the iterative software development process we decided to use in DILIGENT, defines the architecture of a software system as the organization or structure of the system's significant components interacting through interfaces, with components of successive smaller components and interfaces. The Unified Modelling Language (UML) is used to formalize the various elements of the specification and their relationships. Section 2 is fully devoted to present more in detail the concept of software architecture adopted in this document.

In the DILIGENT project, the "smaller components and interfaces", the significant system components are organized in, are identified and explained by the *services specification reports* that are in charge to provide a description of the internal services architecture from a functional, logical, and deployment point of view. To produce the present Deliverable, we relied on these *services specification reports*, i.e. *D1.2.2 DL Creation & Management services specification report*[17], *D1.3.2 Content & Metadata Management services specification report*[18], *D1.4.2 Index & Search services specification report*[19], *D1.5.2 Process Management services specification report*[20], and *D1.6.2 User-Community specific applications specification report*[21].

Having clarified these aspects, this report presents:

- concepts of general interest to the whole system implementation, i.e. Resources, Users, and Virtual Organizations (see Section 3). In particular, the structure and information related to them are reported.
- the integration of the DILIGENT system with the grid infrastructure, i.e. the use of gLite components by the DILIGENT system in order to exploit grid resources (see Section 4).
- the system architecture as a whole(see Section 5). To allow an easy approach to the complexity of the whole system the system architecture is exposed according to the DoW and to the five functional areas identified in [1]. For each of them the UML *architectural view*, that is used to describe the most significant parts of the service from the architectural point of view, is presented. In particular, for each service the components providing interfaces to access the service functionality, the library that can be shared, and the portlets designed are identified and highlighted. Thus, for each service the boundary components are presented, i.e. those components used by the other services to interact with the service black box. The, whereas is possible one or more *deployment scenarios* are used to illustrate possible physical network configurations on which the components can be

deployed and run, and to emphasize details on their level of distribution and replication.

1 INTRODUCTION

The DILIGENT system is a complex system whose engineering is done according to the Unified Process guidelines. As stated by this methodology, the architecture of a complex software system is the organization or structure of the system significant components interacting, through interfaces, with components composed of smaller components and interfaces.

This chapter introduces three main aspects taken into account during the overall architectural specification phase, i.e. the constraints, the technological issues, and the service identification.

1.1 Constraints

When producing an architectural specification of a complex system like DILIGENT, the starting point is the analysis of the constraints we need to take care of. Initially, these constraints usually include general technical project choices. In DILIGENT they are reported in the Description of Work (DoW), as follows:

- the system is based on a *Service Oriented Architecture* (SOA);
- the system exploits Grid technologies;
- the system must reuse existing assets[1] as much as possible;
- the main forming services (or class of services) have already been identified and organized into three logical layers.

Furthermore, the architecture is clearly conceived by considering functional and non-functional requirements, as captured by the D1.1.1 "Test-bed Functional Specification"[1] and by the service specification reports [17], [18], [19], [20], and [21].

After the release of such documents, thank to the new acquired knowledge a new set of further constraints raised. They deal with:

- the environment in which the system/software must operate (mainly the underlying grid middleware/);
- the standards to adopt;
- the compatibility with existing systems;
- the development framework;
- the portlet-based GUIs [12].

Together with the previous ones, all these points have impacted in some way the solutions presented here.

1.2 Technological issues: Service Oriented Architecture and Grid Technologies

Service-oriented architecture (SOA) approach is a way for building distributed systems that deliver application functionality as *services* to either end-user applications or other services. Thus the building blocks of SOA are services. Usually, a service within SOA context is defined as:

"*a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners*"[2].

[1] An asset is intended here as anything (expertise, software) with a potential to earn revenue
[2] What is a Service-Oriented Architecture - http://www.xml.com/lpt/a/ws/2003/09/30/soa.html

The *Web services* technology[3] is the most likely connection technology of service-oriented architectures. It provides a distributed computing approach for integrating heterogeneous applications over Internet making use of open technologies such as XML[3], SOAP[4], UDDI[5], and WSDL[6].

After the first prototype systems, the Grid community understood the limits of any *ad-hoc* solution adopted so far[4]. Therefore, researchers, designers and developers identified the service-oriented approach as the most appropriate paradigm to use in building next generations of grid systems and infrastructures.

This new vision has been formalized in the *Open Grid Service Architecture* (OGSA) [8] where the concept of Grid Service as a network-enabled entity having a well defined semantics in terms of mechanisms for dynamic service creation, lifetime management, notification, manageability, naming and discovering of service instances is the central point.

A further step in this direction was the introduction of *WS-Resource Framework* [10][11] (WSRF) as conventions and proposed standards to enable stateful web services in the OGSA enabling technology that definitely merges the needs of the Web Service and Grid communities.

As a consequence of this evolution, the most appropriate choice to build the DILIGENT system is to design it by relying on - and making use of - the latest outcomes of this process and their related implementations (i.e. technologies that allow to build WSRF compliant services). This decision, of course, strongly impacts on our technological solution. The main consequence is that the DILIGENT nodes forming the infrastructure must be equipped with a *service container* capable to host and support both classical Web services, i.e. services compliant only with basic WS specifications, and WSRF Web services, i.e. services compliant with WSRF specifications. Thus the DILIGENT container must have at least the following characteristics:

1. implementation of *SOAP over HTTP* as a message transport protocol, and both transport-level and message-level security for all communications;
2. implementation of WS-Addressing, WSRF, and WS-Notification standards.

Based on (*i)* these requirements, (*ii)* our best knowledge of services containers, and (*iii)* the deployment model supported, the project adopted the *Java WS Core* [26] as hosting environment for the DILIGENT services.

Further decisions followed this one due to limitations of this technology. In fact, it has no support for the creation of Graphical User Interfaces since it is designed for a service-to-service (S2S) communication and not for direct access by humans. Obviously, DILIGENT has a lot of human interactions and then needs (both in the functional and design phases) to have web portals that provide these types of interactions.

These considerations had the following impacts on the design of the services:

- a selection of an appropriate portal technology was needed;
- the design of a service GUI had to be decoupled from the application logic.

The former was addressed within the WP1.6 services specification report production. A survey of the current technologies was conducted and the portlet-hosting platform was selected as the most suitable for the DILIGENT needs. In a concise manner, Portlets are Web components, managed by a portlet container, that process client requests and generate dynamic content. Portals use portlets as pluggable user interface components that provide a presentation layer.

[3] Extensible Markup Language - http://www.w3.org/XML/
[4] Simple Object Access Protocol - http://www.w3.org/TR/soap/
[5] OASIS UDDI Specification Technical Committee - http://www.oasis-open.org/committees/uddi-spec
[6] Web Service Definition Language - http://www.w3.org/TR/wsdl

The latter conforms to any basic design/implementation approach that suggests to keep the logical layer separated from the presentation layer of an application to avoid a lack of flexibility of the application itself. Therefore, within each technological WP, services have been designed taking care of this point.

1.3 Service identification

Service identification design decisions are essential for SOA success. Just as the proliferation of objects in the object oriented programming hinders the realization of good software, the same is true if services are proliferated. Many early adopters of SOA and Web services quickly realized that the proliferation of Web services does not make for a sound SOA model.

Service identification process in DILIGENT started in the DoW. Here a preliminary set of services was identified and analyzed in order to organize the Workpackages work. In Figure 1 we report the DILIGENT services grouped by conceptual functionality and by logical layers.

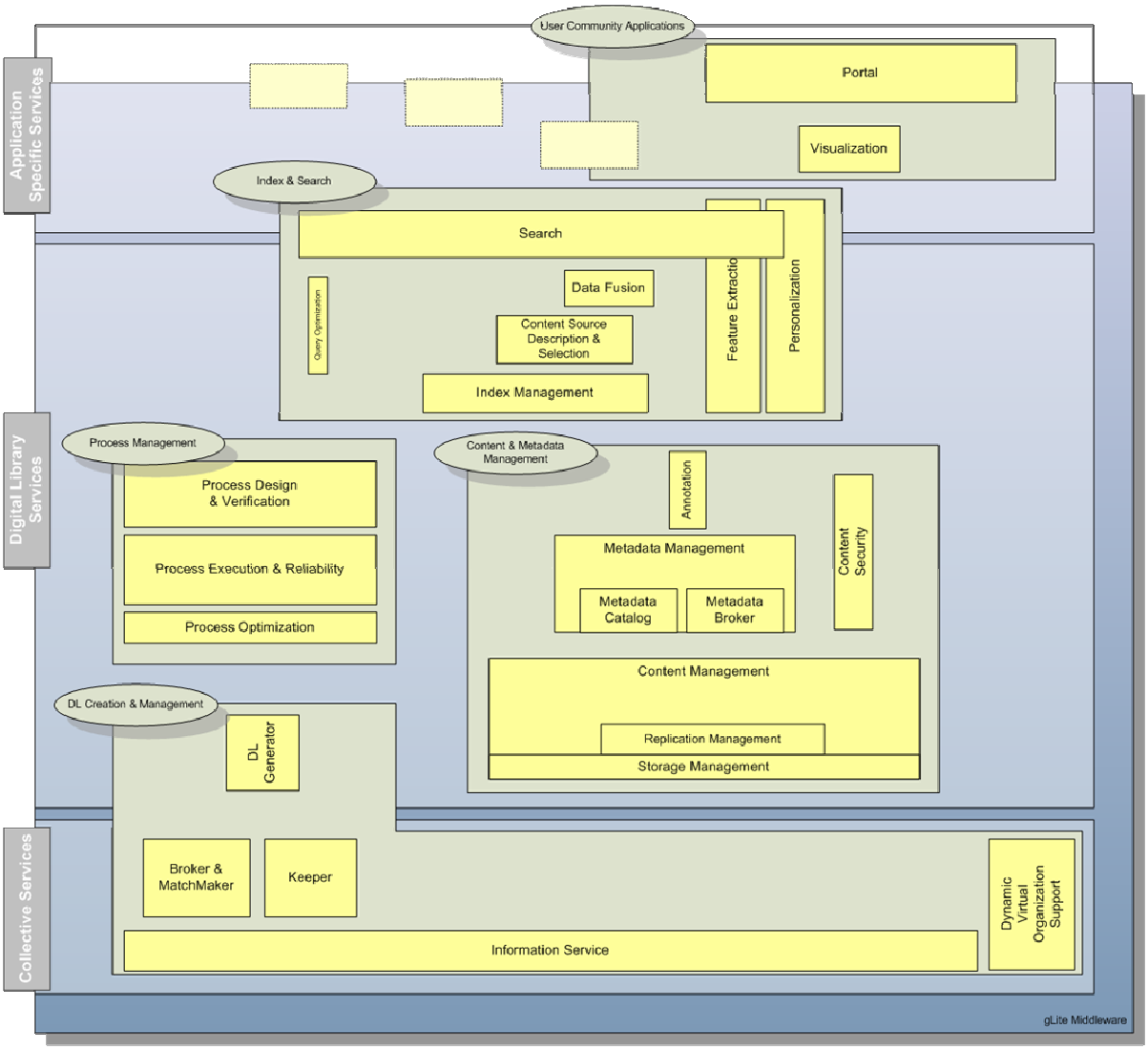


Figure 1. The DILIGENT framework

As stated, this is a *logical* layered architecture which means that it does not model a strict layered system, where the issues are handled at different levels of abstraction and elements of a layer only use elements of the layer below them.

The DILIGENT "layers" are just used to group together functionality related with three main areas: (*i*) services needed for the management of resources (Collective Services), (*ii*)

services offering DL functionalities (Digital Library Services), and (*iii)* services for capturing community specific needs related with the system GUI, Archives and other pre-existing resources (Application Specific Services);

Another point to remark is that each box in figure is not to be necessarily intended as a single entity realized by a unique Web Service. Each of them represents a sort of *meta-service*, i.e. set of related real services that as-a-whole act to supply a certain functionality. Thus, during the design phase, those boxes have been decomposed into many real services by adopting the appropriate architectural pattern [13] that best suits the meta-service functionalities. These decompositions are reported in the D1.X.2 deliverables released by WP1.2-1.6 ([17], [18], [19], [20], and [21]) .

Service identification usually consists of a combination of *top-down, bottom-up, and middle-out* techniques of domain decomposition [14].

The top-down analysis decomposes the domain into its functional areas and subsystems, including its flow or process decomposition into processes, sub-processes, and high-level use cases. In DILIGENT this activity was partially carried out in the DoW and the D1.1.1 "Test-bed Functional Specification"[1], and then reiterated in the service specification reports to obtain fine-grained service decomposition.

The bottom-up approach of the service identification process, is used to analyze and selected existing systems as viable candidates for providing lower cost solutions to the implementation of service functionalities. In some cases, componentization of the legacy systems was needed to re-modularize the existing assets for supporting service functionality. This part of the process is particularly relevant in DILIGENT because the project aims at integrating pre-existing technologies as much as possible.

The middle-out view consists of goal-service modeling (i.e. identifying the goals and sub-goals that must be realized in order to support higher level goals, then associating the sub-goals with the services required to realize them) to validate and discover other services not captured by either top-down or bottom-up service identification approaches.

1.4 Document Outline

The rest of this document is structured as follows. Section 2 explains the rationale used in producing the document. Section 3 presents the Resource, User and Virtual Organization models on which the DILIGENT architecture relies on. Section 4 reports on the integration of DILIGENT with gLite, i.e. the grid middleware released by the EGEE project. Section 5 presents an integrated and comprehensive view of the DILIGENT system architecture. Finally, section 6 concludes this deliverable.

2 RATIONALE OF ARCHITECTURAL SPECIFICATION

This Architectural Specification presents the software architecture of the DILIGENT system. In this section we present the rationale that has driven the production of this specification and introduce the context in which we are acting by discussing what is and what is not a software architecture, and what we are going to define.

2.1.1 What is software architecture?

Software architecture is the high-level structure of a software system. Important characteristics include:

- it is at a level of abstraction high-enough to make the system visible as a whole;
- its structure must support the required system functionality. Thus, the dynamic behavior of the system must be taken into account when designing the architecture;
- its structure must conform to the system qualities (also known as non-functional requirements). These at least include performance, security and reliability requirements associated with current functionality, as well as flexibility or extensibility requirements associated with accommodating future functionality at a reasonable cost of change. These requirements may conflict, and tradeoffs among alternatives are an essential part of designing an architecture;
- at the architectural level, most of the implementation details (mainly the internal structure of the components, we explain them later on) are hidden, although general technologies that influence the architecture of the system are faced.

Software architecture is commonly defined in terms of *components* and *connectors*. Components are identified and charged with responsibilities and component clients interact them through "contracted" interfaces. Component interconnections (i.e. connectors) specify communication and control mechanisms, and support all component interactions needed to accomplish system behavior.

In creating the software architecture, we are concerned with:

- *meta-architecture*: the architectural vision, style, principles, key communications and control mechanisms, and concepts that guide the team of architects in the creation of the architecture.
- *architectural views*: complementary models or views to focus on and capture certain aspects of the system. In particular, logical views help to document and communicate the architecture in terms of the components and their relationships; these models are useful in thinking through how the different entities interact to accomplish their assigned responsibilities and evaluating the impact of what-if scenarios on the architecture. Deployment views help in evaluating physical distribution options, and documenting and communicating decisions about the distribution scenarios.
- *architectural patterns*: structural patterns such as SOA layers, client/server and WSRF Factories, and mechanisms such as brokers and bridges.
- *key architectural design principles* including abstraction, separation of concerns, postponing decisions, simplicity, and related techniques such as interface hiding and encapsulation.
- system decomposition principles and good interface design.

The architecture supplies the blueprint for the implementation teams because:

1. it defines how to assign the work that must be carried out;
2. it is the carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision;

3. it is a vehicle for early analysis to assure that the design approach will yield an acceptable system;
4. it is the artifact that holds the key to post-deployment system understanding efforts.

In short, architecture is the conceptual glue that holds every phase of the project together.

An obvious truth is that the most perfect architecture is useless if it is not understood or (perhaps worse) misunderstood. Documenting the architecture is the critical, crowning step to crafting the architecture.

2.1.2 What software architecture is not

Software architecture must be distinguished from (i) service (or system) design and implementation, and (ii) from other types of related architectures. For instance, software architecture is not the information (or data) model, though it uses the information model to get type information for method signatures or interfaces. Further, it is not the architecture of the physical system, including processors, networks, and the like, on which the software will run. However, it uses this information in evaluating the impact of architectural choices on system qualities such as performance and reliability. More obviously, perhaps, it is not the hardware architecture of a product to be manufactured. While each of these other architectures typically has its own specialists leading their design, these architectures impact on and are impacted by the software architecture, and where possible, should not be designed in isolation from one another. This is the domain of *system* architecting.

2.1.3 System Design versus System Architecture

The terms "system architecture" and "system design" are often used interchangeably; actually, they deal with two very different engineering processes.

In approaching the distinct but related tasks of architecture and design, it is important to remember these key principles:

- the system architecture focuses primarily on the overall structure of a system, identifying major components and their relationships and how they should be organized with respect to each other;
- the system design determines how the desired functionality should be realized by the system implementation.

Thus, the architect focuses on issues such as which components should communicate with one another, which should be visible to one another, and which should be replicated; how components should be distributed, how they should talk to one another, and where they should be stored.

In the DILIGENT context, the system design is captured by the D1.X.2 "Service specification" deliverables ([17], [18], [19], [20], and [21]).

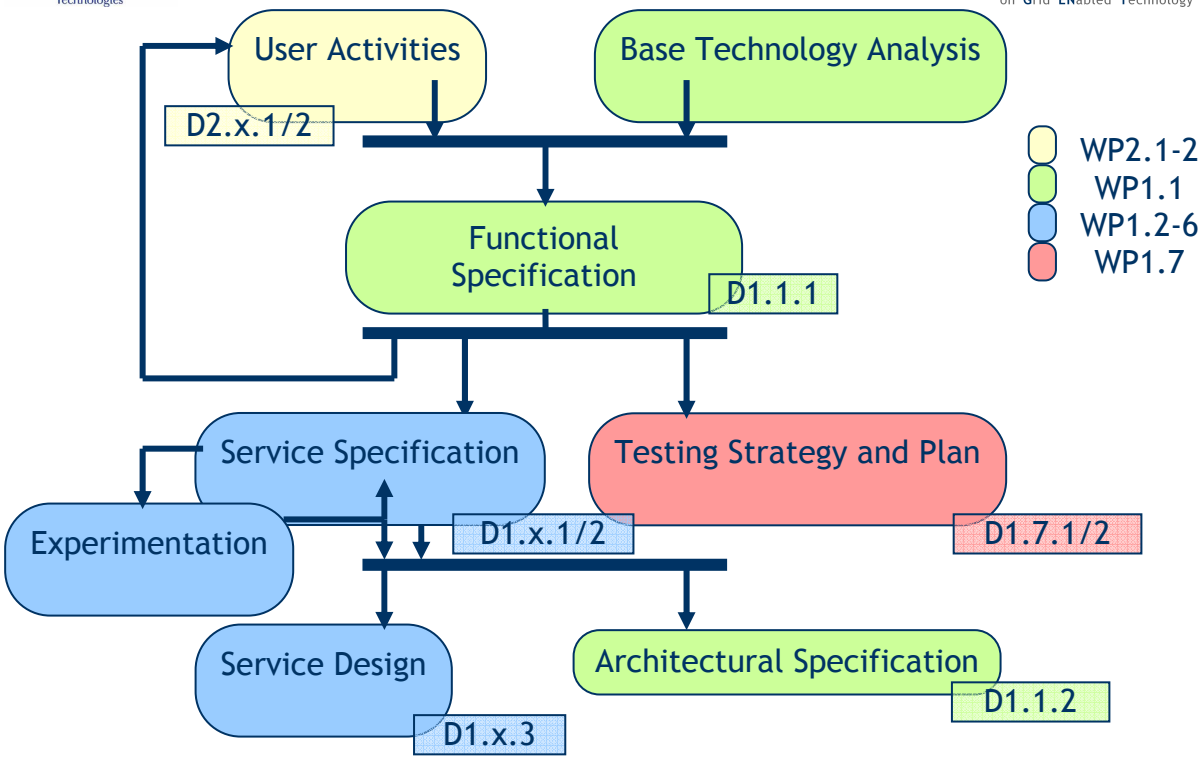


Figure 2. Relationships among activities and deliverables

Figure 2 depicts the relationships among other deliverables and the deliverable D1.1.2.

3 RESOURCES, USERS AND VIRTUAL ORGANIZATIONS

The main goal of the DILIGENT project is to exploit the Grid technology to create on-demand/dynamic Virtual Digital Libraries (VDLs) i.e. Digital Libraries that (*i*) are dynamically built by aggregating a pool of *resources*, (*ii*) provide access to an appropriate pool of *users* to such resources, and (*iii*) optimize the resources usage in accordance to the sharing policies by relying on the highly controlled *virtual organization* (VO) sharing mechanism.

Thus, the first step the project addressed was the clear definition of the following issues: what resources DILIGENT manages, users profiling and how they are organized in VOs.

In this section such orthogonal concepts are presented to create a common background for the whole design.

3.1 DILIGENT Resources

The DILIGENT Resources are the "entities" that the DILIGENT system manages for the correct handling of Virtual Digital Libraries. They must be defined regardless the implementation solutions adopted.

A DILIGENT Resource is anything whose related information can be gathered, stored, monitored, and disseminated in order to provide the valuable amount of knowledge needed during the creation and management of a VDL as well as to operate the entire DILIGENT infrastructure. Therefore, such resources are not only an abstract idea, rather they need a concrete realization in the developed solution. Without them, it is not possible to achieve the goals that DILIGENT promises. It is worth noting that resources of different nature have different physical manifestations.

In the following, we introduce the DILIGENT Resources by drawing on the MDA (Model-Driven Architecture) framework [15], an approach that uses models in software development, as proposed by the OMG consortium.
The core idea we get from MDA is the usage of two viewpoints:

1) a *Platform Independent Viewpoint* that shows that part of the complete specification that does not change from one platform to another; therefore it includes no detail about the implementation platform. This viewpoint is represented by a set of models. In the MDA Guide [16] a model is defined as follows: "*A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language.*". Globally, the aim of the models is to provide a Platform Model, i.e. "*a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform*". Models are formalized using UML class diagrams.
2) a *Platform Specific Viewpoint* that combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system. This viewpoint is represented by the information (named *resource profile*) that must be associated to each resource to discover this resource in the system. We describe the resource profiles using the XML/XML Schema in Section 3.1.2.

The design of the services dealing with the Digital Library creation and management (see Section 5 and [17]) takes strongly care of the DILIGENT Resources. A dedicated Registration Service maintains an entry for each of them and a registration procedure must be followed to create these entries. As we will see later on, the registration can be performed in *automatic mode* (by other services) or *request mode* (by authorized users) via the Registration Service (see section 5.1.4.5). One of the basic steps of such a procedure is that the DILIGENT Resource owner (named *Resource Manager*, see Section 3.2) must provide the profile for each new resource. Then, this profile is disseminated via the DIS

services (see Section 5.1.1) to allow other services to discover and to provide an access point for the DILIGENT Resources.

The goal of the described functionality is to achieve a result similar to that obtained with the UDDI (Universal Description, Discovery, and Integration) based mechanism in the Web Service world. UDDI provides a method for publishing and finding service descriptions. In the UDDI context, the "profile" of the Web Service is represented by the WSDL service interface published in the UDDI Registry. The difference from the UDDI mechanism is that there the goal is to set up a service-to-service communication, while in DILIGENT we aim at connecting a DILIGENT Resource with its consumers.

Profile vs State

Profiles are only a subset of the information captured by the Models presented in Section 3.1.1. Each resource has also a *state* that includes the dynamic information related to the activity performed during the resource lifetime. The DILIGENT service that manages the DILIGENT Resource is in charge of exporting their state. For instance, the HNM service (see Section 5.1.3) is in charge of exposing the state of a DHN it manages (including ProcessorLoad, StorageDevice availability, list of DILIGENT packages - i.e. WSRF services, libraries, portlets, grid jobs - deployed, etc.). Having a look at the implementation of the system, the states are exposed as WS-ResourceProperties documents.

3.1.1 Resource Model

The goal of the Resource Model presented in Figure 3 is to capture the structure and the main relationships among the resources.

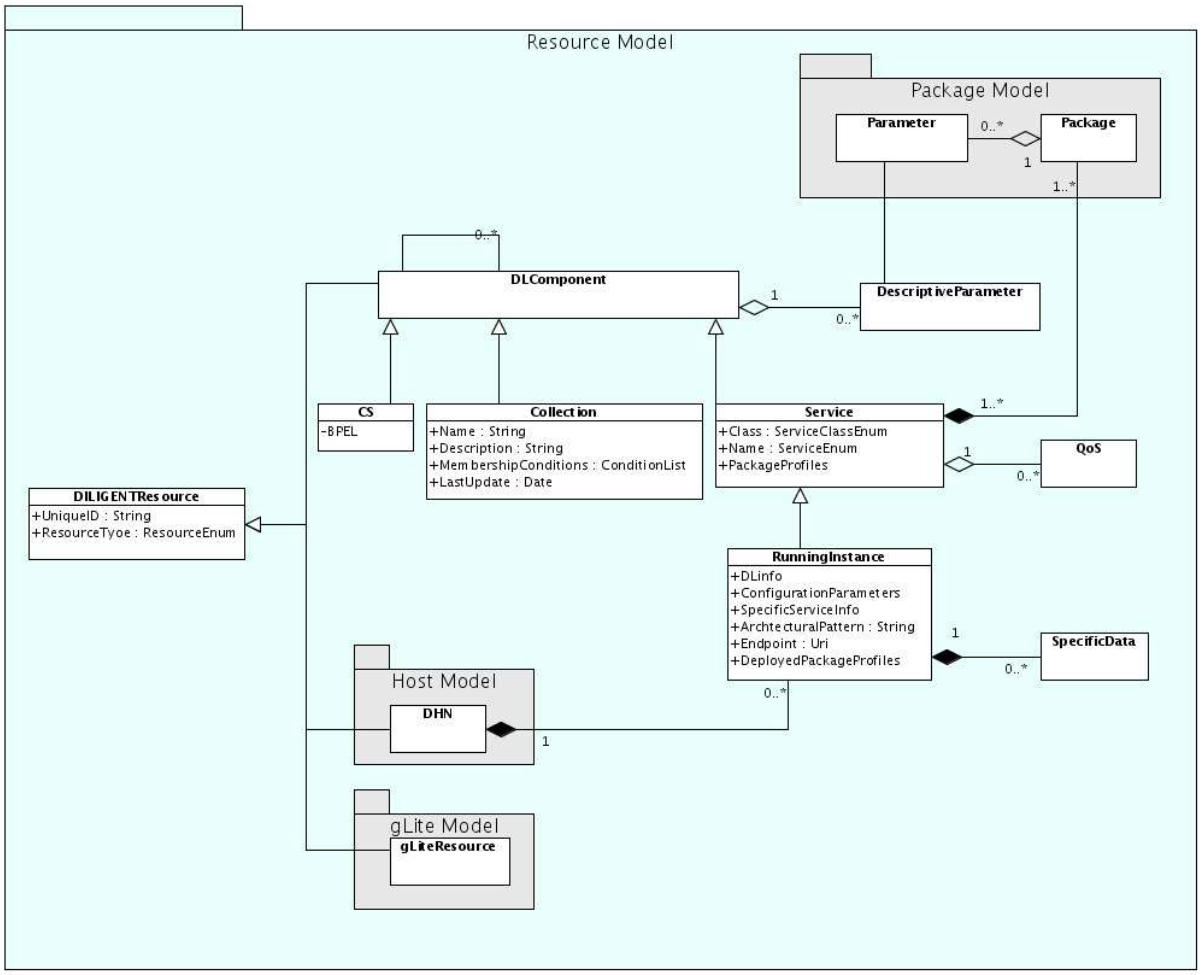


Figure 3. The DILIGENT Resource Model

3.1.1.1 The Resource Model Entities

The root class, *DILIGENTResource,* models the generic DILIGENT Resource. It is characterized by a unique identifier, that identifies it unambiguously, and by a type, that discriminates among the different resource types. By specializing this generic concept of resource, the following types of DILIGENT resources have been identified:

DHN *(DILIGENT Hosting Node)*

This class models a hosting node, i.e. a computer machine connected to the network, available within the DILIGENT infrastructure and capable to host Packages and to execute CSs. A description of this DILIGENT Resource is reported in 3.1.1.5

gLiteResource

gLiteResource models the resources of the underlying grid infrastructure that can be monitored and used by the DILIGENT services. They are discussed in Sections 3.1.1.4 and 4.

DLComponent

The *DLComponent* resource models the bricks that can be selected during the definition of a VDL and that are then dynamically composed to create the VDL. The information that characterizes a *DLComponent* is:

- a type, to discriminate among the different components that can be used to build a DL: Collection, Service, and CompoundService (CS)[7].
- a list of attributes, to specify descriptive features of the component (when the *DLComponent* is a Service, these attributes are directly associated with the Parameters of the Packages that compose the Service);
- a list of relationships with other *DLComponents* to express dependencies among them.

The way in which DLComponents have to be described in terms of attributes and relationships as well as the capability to express constraints on their composition (so that they can be used by the VDL Generator Service - see Section 5.1.5) needs further in-depth investigations. This is expected in the D1.2.3 report.

Service

A *Service* is a software system that delivers functionalities. In DILIGENT, a *Service* is composed by a not-empty set of related *Packages* (identified through dependencies) forming an unique logical entity. A *Package* (in the rest of this document we also refer it as *software package*) is the smallest installable unit of software that can be deployed on a DHN. *Packages* are the way in which the software needed to set up a DL prepared in order to be used and stored in the system. The set of *Packages* forming a *Service* is composed by:

1) one and only one *Package* of WSRFService[8] or Portlet type (the main package or the entry point)
2) an arbitrary number of other *Packages* (Library and GridJob) used by such a *Package*.

It is worth noting that *Services* are annotated with information reporting on the Quality of Service (QoS) they provide. The semantic of this information as well the attributes captured are different in accordance with the different nature of services. Instances of attributes are:

- *Availability*, i.e. the probability that a resource can respond to requests;

[7] This list can be extended to cover unforeseen needs
[8] Package types are discussed in Section 3.1.1.3

- *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 an initial set that will be enriched with further aspects, specific of particular resources within the D1.2.3 deliverable. 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 point is related with the "quality" of the parameter advertised by the resource, e.g. a certified attribute states the "authority" responsible for expressing the attribute value, a type attribute discriminates between objective measurements automatically taken and subjective measurements expressed by humans.

RunningInstance

A `RunningInstance` is a specialization of the `Service` class, i.e. it models an active `Service`. Therefore, it is composed by a set of Packages deployed and running. The "main" Package of such a set is the WSRFService or the Portlet and most of the `RunningInstance` properties are related to this Package. This is because such a Package is the access point of the `RunningInstance` functionalities, i.e. other `RunningInstances` cannot contact directly a deployed Library or a GridJob of a remote `RunningInstance`, rather they can interact with its WSRFService or Portlet.

A `RunningInstance` is always associated to one DHN in the same way as a deployed WSRFService/Portlet runs only on one node of the infrastructure.

CS *(Compound Service)*

An exciting possibility emerging from the interoperable and flexible nature of a SOA is the capability to recombine Services into more powerful and complex workflows, known as *Compound Services* (`CS`) or *Processes*. Because a `CS` is a combination of independent services, the activities of the process (i.e. the component services forming the `CS`) can be executed on different DHNs. This resource models a `CS` specification [20].

Collection

A `Collection` is a resource modeling a portion of the Digital Library information space, i.e. the set of information objects[9] available within a DL. In detail, a `Collection` is an aggregation of related information objects having explicit relationships with other information objects. [18].

3.1.1.2 The Resource Model Entities Relationships

Regarding the relationships, we can appreciate that:

- one `DLComponent` can have one or more `DescriptiveParameters` (used at VDL definition time by the VDL Generator service)
- one `DLComponent` can be associated with another `DLComponent` to express dependencies among them
- one `Service` is composed by one or more `Packages`
- one `RunningInstance` is always associated with one DHN
- one DHN is composed by zero or more `RunningInstances`
- one Package can have one or more Parameters (used at DL creation time by the Keeper and Broker & Matchmaker services)

[9] for a detailed explanation about information objects, see Section 5.2

3.1.1.3 Package Model

The Package Model is that part of the Resource Model that is focused on *Packages*. As stated, concretely, a Package is a "piece of software" that can be deployed on a DHN. A *Package* can be:

- a *WSRFService*, representing a package that once deployed produces a WSRF Service (a service able to manage stateful resources following the WS-Resource patterns);
- a *Portlet*, representing a package that once deployed produces a portlet (Web components, managed by a portlet container, that process requests and generate dynamic GUI content) that can be hosted by the DILIGENT Portal;
- a *GridJob*, representing a package containing the code and the related information needed to run a certain job on the grid infrastructure (see Section 4.1.3);
- a *Library*, representing a software library that can be hosted on a DHN and is needed by other packages to support their tasks. There are two types of libraries:
    1. *Shared Library*, software library offering functionality of common utility, e.g. an XML parser library or a mathematical support library;
    2. *Stub Library*, software library offering functionality for interacting with WSRF Services.

The Package Model is illustrated in Figure 4.

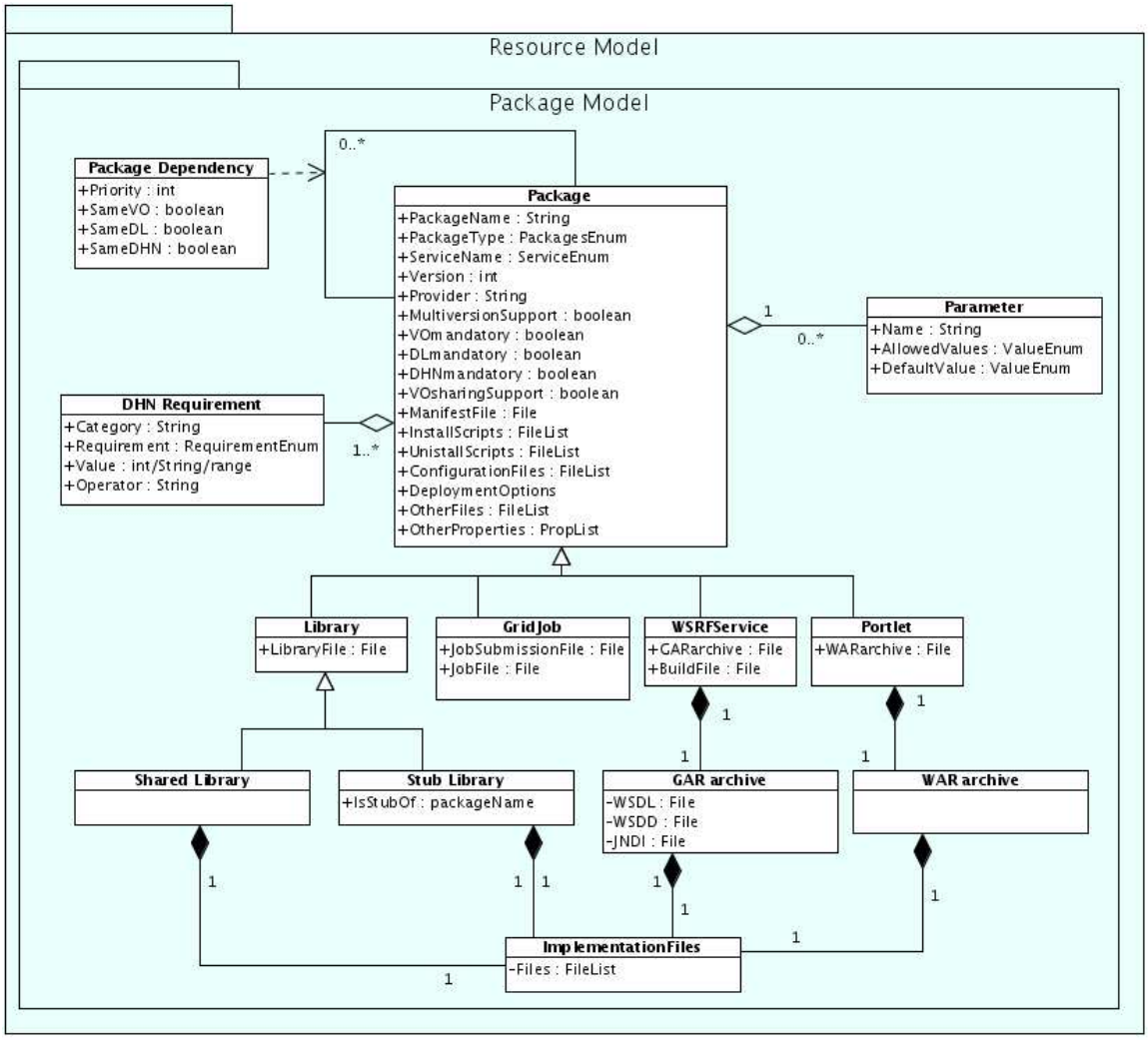


Figure 4. Package Model

The Package structure

Despite its definition (a piece of software), a `Package` is not only a bunch of binary files for installing a software (e.g. a GAR archive for a WSRFService or a JAR archive for a Java Library), rather it is a collection of related files and information that guarantee the correct management of the package in all its phases (storage, moving, deployment, configuration, activation and monitoring). We discuss here the files that can be included in a `Package`, while the information is discussed in Section 3.1.2.2 since it is part of the `Service` profile.

A `Package` is registered in the Registration Service, as part of the procedure for registering a `Service`. That is, for each Service, a *Resource Manager* (see Section 3.2), i.e. the owner of the service resource, must upload the `Packages` that compose that `Service`. A `Package` must include the following files:

- an *archive file* (packaged following the specific format for the type of that package, i.e. GAR, JAR, WAR, or executable) with the implementation files;
- *install/uninstall scripts*, automatically executed on the target DHN by the system before and after the deployment/un-deployment of the `Package`; they can be used to execute specific tasks that prepare the environment for the execution of the software(third party software installations, environment variables, creation of a particular structure in the file system, checking of dependencies, etc.);
- alternative *configuration files;*
- a *manifest file*, i.e. a file containing additional information about the `Package` as a whole as well as an entry for each component file. Examples of such information are the 'created-by', the class-path, the main-class, the content-type;
- *anything* needed for the correct execution of the software; all these additional files are moved together with the package on the target DHN in a special folder, where the software or the scripts included in the package can access them.
- *License information*

How to organize the `Package` structure is still under investigation within the WP1.2 at the time of writing this document.

3.1.1.4 gLiteResource Model

The gLiteResource Model focuses on the DILIGENT Resources created to interact with the underlying grid infrastructure. The information related with this type of DILIGENT Resource is mainly obtained by gathering the data that these resources advertise, and thus it strongly depends on their design. These data are harvested from the R-GMA Server(s) of the gLite-based Grid infrastructure (see Section 4) by the DIS-R-GMAClient (see Section 5.1.1.1). The R-GMA Server exposes the information about the gLite services, ComputingElement and StorageElement following the GLUE Schema definition [22].

Figure 5 illustrates the Model with the structure and the main characteristics of the information about the `gLiteResources` maintained and disseminated by the DIS services.

This part of the Resource Model is subject to change due to the evolution of gLite middleware.

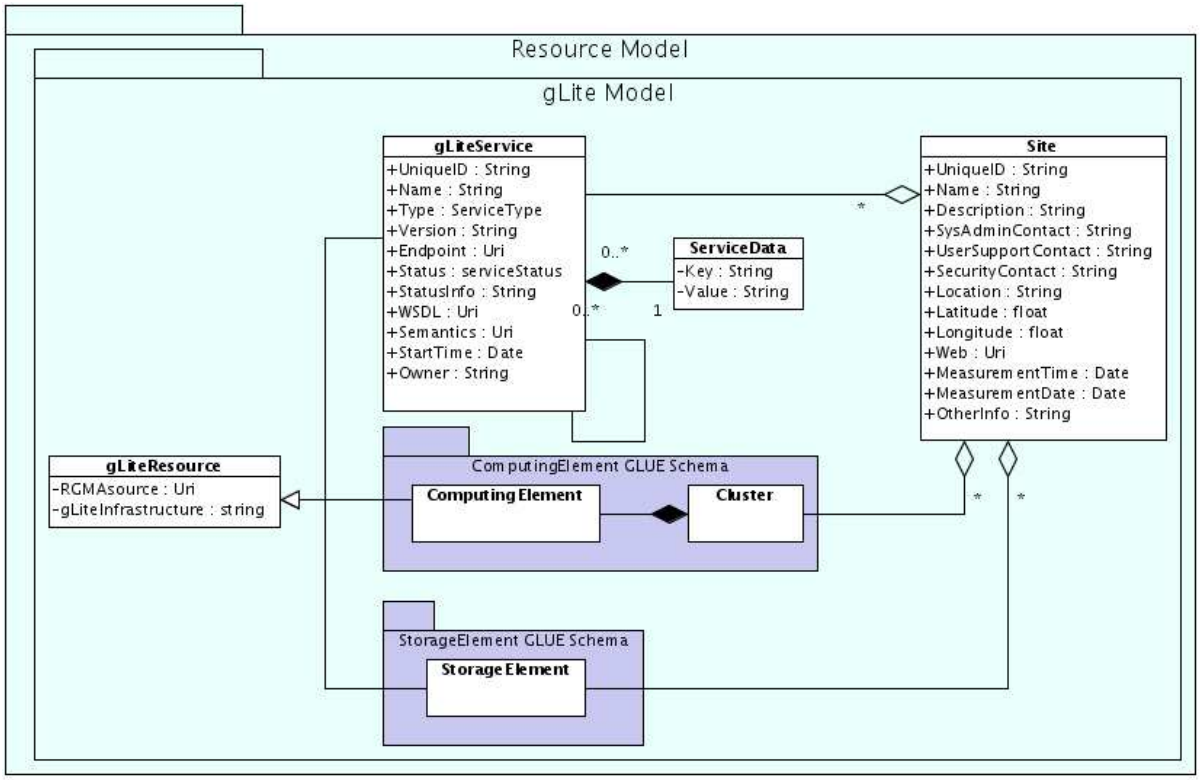


Figure 5. gLite Resource Model

GLUE aims at defining an information model and the mapping of the status information into concrete schemas for representing Grid resources. The GLUE Schemas for CE and SE (extracted from [22]) are reported in Figure 6 and Figure 7.

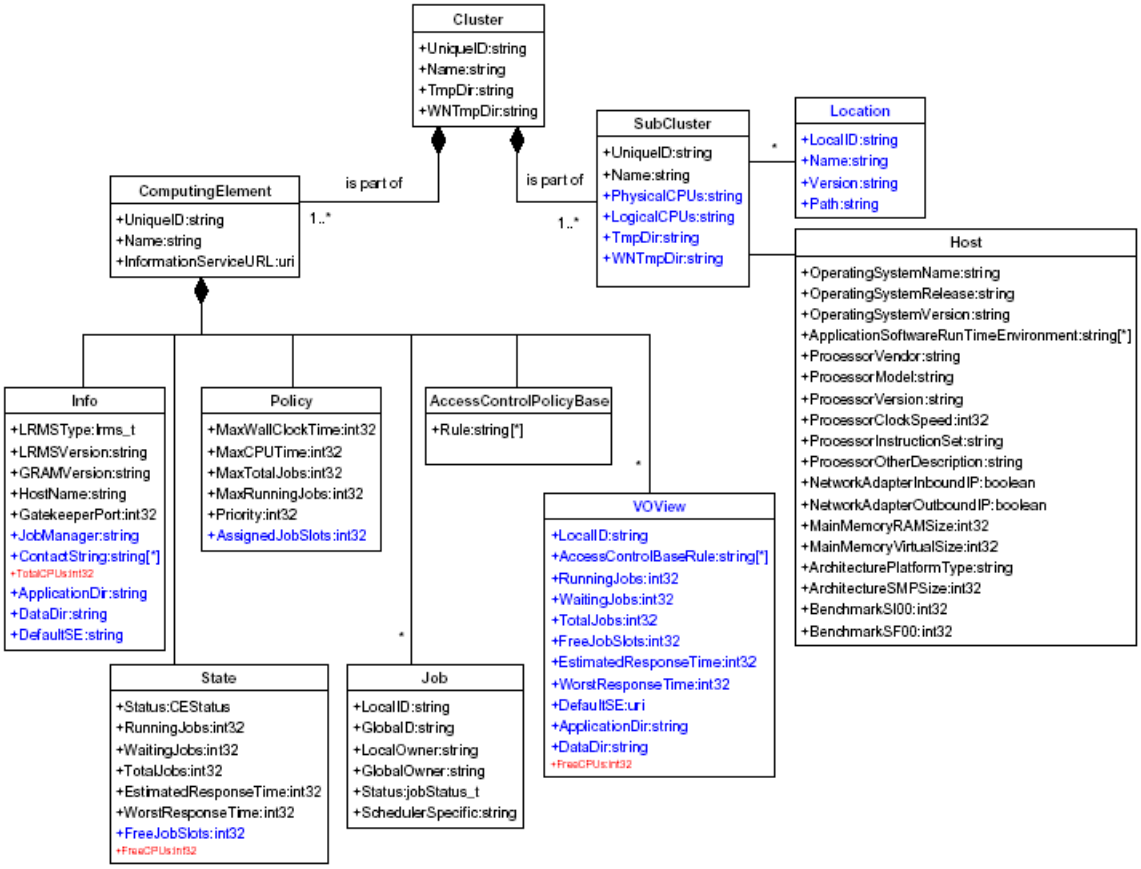


Figure 6. CE as represented in the GLUE Schema

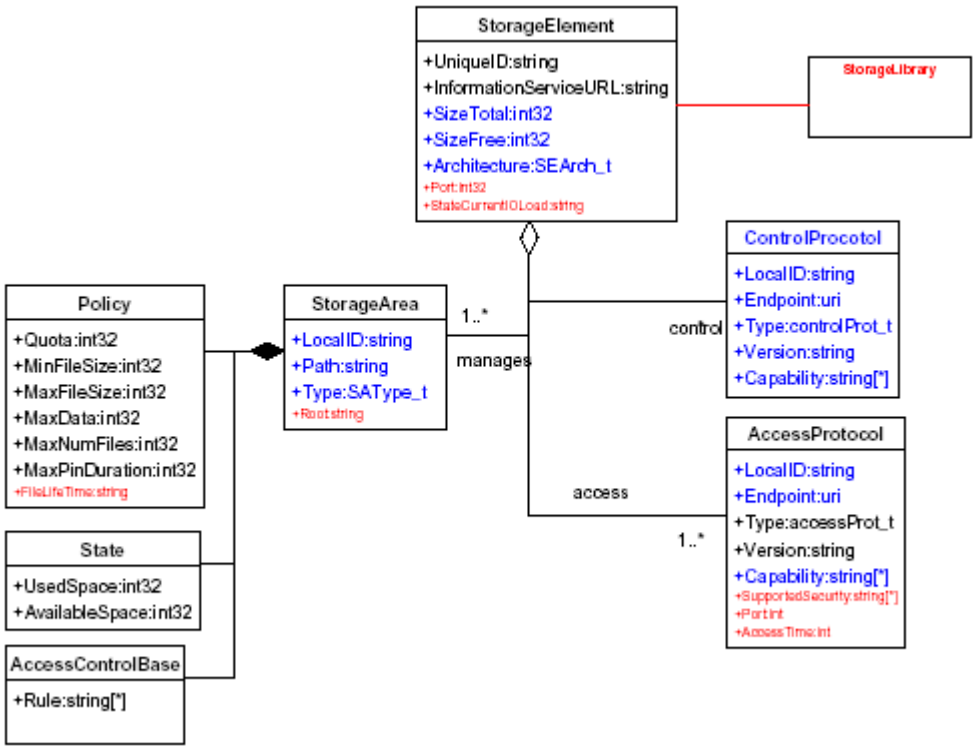


Figure 7. SE as represented in the GLUE Schema

3.1.1.5 Host Model

The Host Model is defined within the GLUE Schema Specification [22]. The kind of resource this schema takes care of is the node where DILIGENT services are deployed and run, i.e. what we have called DHN. The Model's main goal is to serve as a support for selecting the best DHN on which:

- to deploy a RunningInstance, a task performed by the Broker & Matchmaker Service (see Section 5.1.2), or
- to run a CS or a part of it, a task performed by the Process Management Service (see Section 5.4).

The following diagram shows the Host Model as from the GLUE Specification:

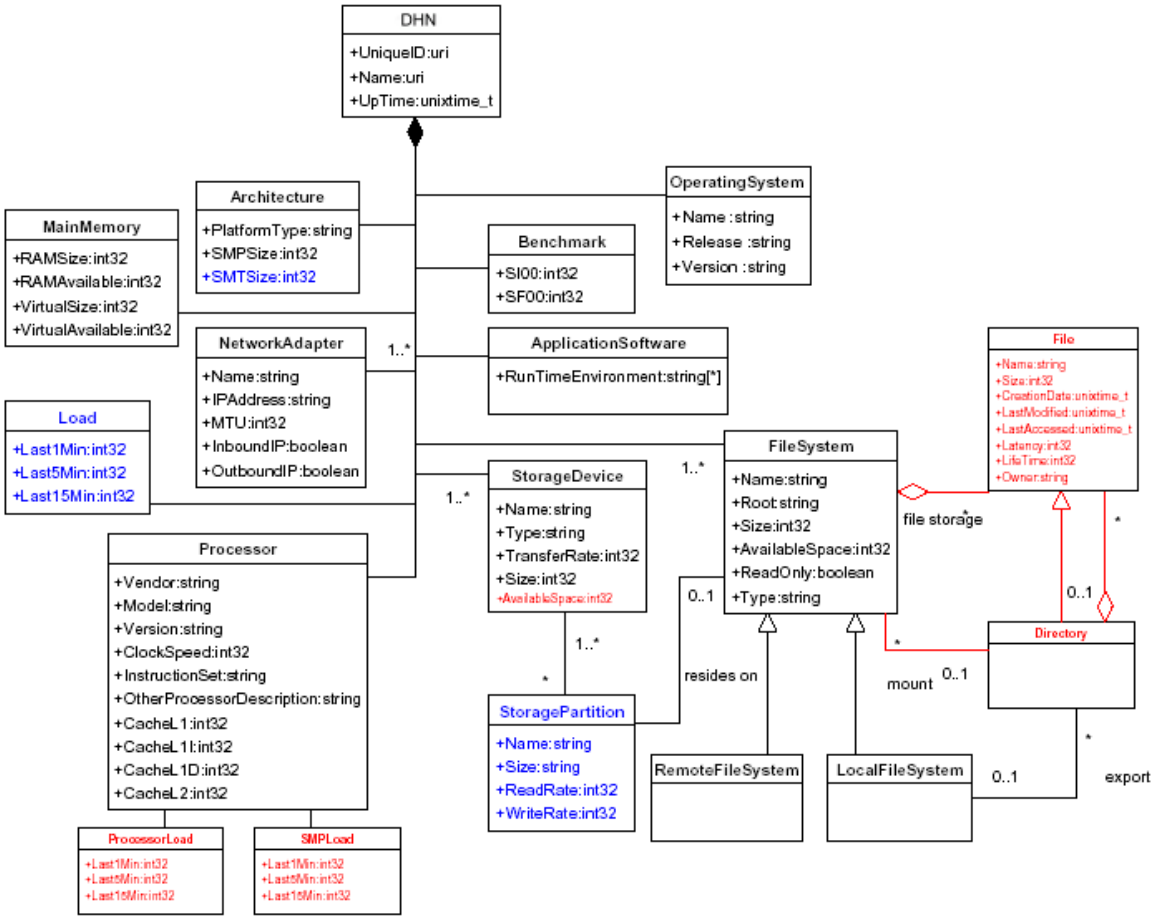


Figure 8. Generic Host Model as represented in the GLUE Schema

The above model is not complete for our purposes since it lacks of a description of installed DILIGENT Packages that compose the RunningInstances hosted by the DHN; this is indeed required to properly satisfy matchmaking requests. For this reason, we enriched it by modeling the set of packages actually installed on a DHN. This extension comes from the relationship between the DHN and the RunningInstance resource reported in the Resource Model. The extension is depicted in Figure 9.

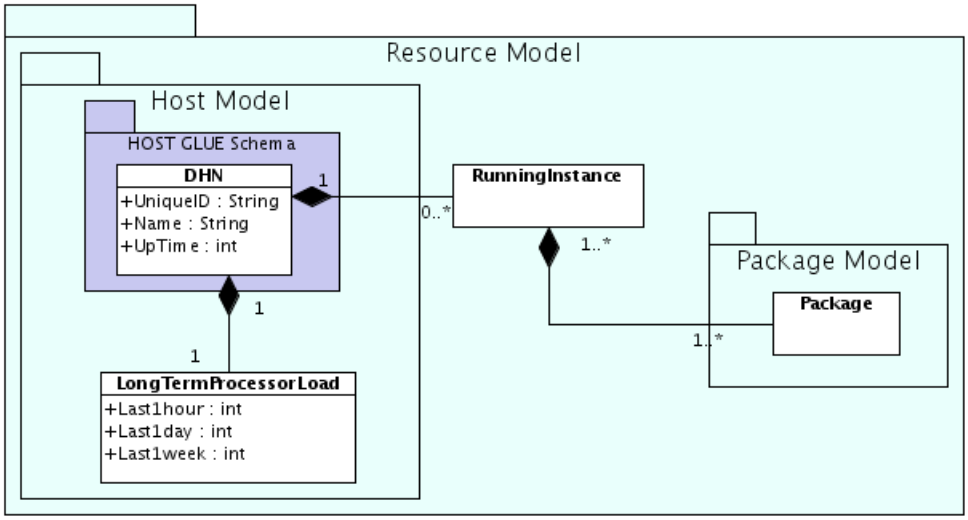


Figure 9. Extended GLUE Host Model

A further modification of the original GLUE Schema is related to the `ProcessorLoad` category. It is also to be noticed that at the time `last-hour`, `last-day` and `last-week` average load properties are also maintained for each host in the `LongTermProcessorLoad` class.

The reason for this extension is that `RunningInstances` are usually hosted on a `DHN` for a relatively long period of time. When deploying a `Package` of a `RunningInstance` it is useful to know long-term processor load instead of very short-term load as the `last-min`, `last-five` and `last-fifteen` measures are.

3.1.2 Resource Profile

As stated, the profile is the set of information related to each DILIGENT Resource supplied at registration time. Its two-fold purpose is:

- to support resource discovering
- to store information that helps to handle the resource correctly

Clearly, each Resource type has its own profile structure.

Physically, profiles are described as XML files compliant with the following XML Schema:

XML Schema for DILIGENT Resource profile

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
   <xs:element name="UniqueID" type="xs:string"/><!-- assigned by the
RegistrationService-->
   <xs:element name="ResourceType">
      <xs:simpleType>
         <xs:restriction >
            <xs:enumeration value="Service"/>
            <xs:enumeration value="CS"/>
            <xs:enumeration value="Collection"/>
            <xs:enumeration value="RunningInstance"/>
            <xs:enumeration value="DHN"/>
            <xs:enumeration value="gLiteResource"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <xs:element name="AuthorizationPolicies">
      <xs:complexType/>
   </xs:element>
   <xs:element name="Profile">
```

```
      <xs:complexType>
        <xs:sequence>
          <xs:any/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  <xs:element name="DILIGENTResource">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="UniqueID"/>
        <xs:element ref="ResourceType"/>
        <xs:element ref="AuthorizationPolicies"/>
        <xs:element ref="Profile"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The Profile element identifies a resource-specific section that includes the information described in the following. For each resource a possible example of such a section is reported. It is important to notice that such examples are introduced here just to clarify the concepts, but they are subject to change in the next project phases.

3.1.2.1 DLComponent profile

DLComponent is only an abstract resource used to group the DILIGENT Resources used to define a VDL. Therefore, it does not have a specific profile.

3.1.2.2 Service profile

The Service profile must include:

- the class of the service, that identifies the service role within the Digital Library. Allowed categories are: Search, Index, MetadataManagement, ContentManagement, Portal, DataFusion, Annotation, CSD, CSS, ProcessManagement, FeatureExtraction, Personalisation, ContentSecurity;
- the service name;
- the descriptive parameters that make it possible to interpret and configure the Service (i.e. its RunningInstance) at deployment time;
- the list of packages needed to deploy the service on DHNs.

For each package there must be a section for expressing a set of information compliant with the Package Model presented in Figure 4. Namely:

- descriptive information like name, type, version, provider, classification w.r.t Package and Resource Models (e.g. PackageType, ServiceName);
- deployment information that needs to be validated by the DILIGENT system. This second type of information can be further divided into two subtypes:
    - *Package dependencies*, expressing the need of a package to be deployed together with other packages in order to work properly;
    - *DHN requirements*, expressing the requirements the Broker & Matchmaker uses to choose the DHN where the package should be deployed;
    - *mandatory labels*, forcing the deployment of the package in some special situations;
    - *configuration parameters* supported by the package.
- package properties, a group of information that describes the static properties of the package once it is deployed.

All this information is mainly used by the Collective Layer services to "inject intelligence" into packages handling in various stages; some parts are also included in the profile of the

RunningInstance dynamically created by the services of the Collective Layer in charge of deploying and creating the service instance.

Service profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<DILIGENTResource>
   <UniqueID>
      <!-- to be assigned by the RegistrationService-->
   </UniqueID>
   <ResourceType>Service</ResourceType>
   <AuthorizationPolicies/>
   <Profile>
      <Class>Search</Class>
      <Name>QueryParser</Name>
      <DescriptiveParameters>
         <param>
            <!-- to be defined -->
         </param>
         <param>
            <!-- to be defined -->
         </param>
      </DescriptiveParameters>
      <QoS>
         <!-- to be defined -->
      </QoS>
      <DLDependencies>
         <DLComponent>
            <Class>Search</Class>
            <Name>QueryParser</Name>
            <DescriptiveParametersValue>
               <param>
                  <!-- to be defined -->
               </param>
               <param>
                  <!-- to be defined -->
               </param>
            </DescriptiveParametersValue>
         </DLComponent>
      </DLDependencies>
      <PackagesList>
         <Package>
            <PackageName>Parser</PackageName>
            <PackageType>WSRFService</PackageType>
            <Version>1.0</Version>
            <DLMandatory value="1"/>
            <VOMandatory value="0"/>
            <DHNMandatory value="0"/>
            <DisposeInterfaceSupport value="1"/>
            <MultiVersionSupport value="0"/>
            <VOSharingSupport value="1"/>
            <ManifestFile>file1</ManifestFile>
            <InstallScripts>
               <file>file2</file>
               <file>file3</file>
            </InstallScripts>
            <UninstallScripts>
               <file>file4</file>
               <file>file5</file>
            </UninstallScripts>
            <Parameters>
               <param name="param1">
                  <allowed_values>
                     <value default="1">value1</value>
                     <value>value2</value>
                  </allowed_values>
               </param>
            </Parameters>
            <Dependencies>
```

```
            <Package>
               <PackageName>Index&amp;amp;SearchLibrary </PackageName>
               <SameDHN value="1"/>
               <Priority value="1"/>
            </Package>
         </Dependencies>
         <DHNRequirements/>
         <ConfigurationFiles>
            <file>config_file1</file>
         </ConfigurationFiles>
         <WSRFService>
            <GARArchive>Parser.gar</GARArchive>
            <BuildFile>build.xml</BuildFile>
            <DeploymentOptions>-Dprofile config_file1 –debug
            </DeploymentOptions>
            <WSDL>
               <!-- WSDL file content go here....-->
               <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
               <xs:import
schemaLocation="http://www.w3.org/2006/01/wsdl/wsdl20.xsd"
               namespace=""/>
            </xs:schema>
            </WSDL>
         </WSRFService>
         <OtherFiles/>
         <xs:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xs:element name="OtherProperties">
               <xs:any minOccurs="0" maxOccurs="unbounded"/>
            </xs:element>
         </xs:schema>
      </Package>
   </PackagesList>
</Profile>
</DILIGENTResource>
```

3.1.2.3 RunningInstance profile

The RunningInstance profile reports the information that allows other DILIGENT services to automatically discover and, thus, interact with this instance. The registration of such a profile is performed by the services that dynamically deploy the Service and activate the RunningInstance (specifically, the HNM service, see Section 5.1.3). Some data are added at deployment time (such as the URI) by the HNM, while others are extracted from the Service profile.

The profile of a RunningInstance includes:

- the Service profile the RunningInstance specializes;
- the data of the DLs this instance joins
- some deployment information
    - among the others, this section includes also the Serialized State information; this is a Logical File Name of a file stored on the grid which is used to reconfigure the RunningInstance with the state of a previous instance of the Service. When a instance wants to save its state for future uses, it must serialize such a information on the grid and store the LFN used on its profile. Then, when a new instance is a replica or a substitute of another instance, the LFN in the Serialized State section must be used by the new instance to recover the state of the other one. For instance, the Package Repository (see Section 5.1.3) serializes the table including the mapping between the package names and their physical locations, if a new Package Repository is instantiated the new instance recovers these data, rebuilds the state and is able to serve the requests as the other instance;
- information on which architectural pattern has been adopted (singleton or factory);
- the factory service URI and/or the singleton service URI;

- values of the configured parameters;
- service specific data. This is a sort of "open section" where a service can put any runtime information that helps to select a particular instance among others.

RunningInstance profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<DILIGENTResource>
   <UniqueID>
      <!-- to be assigned by the RegistrationService-->
   </UniqueID>
   <ResourceType> RunningInstance</ResourceType>
   <AuthorizationPolicies/>
   <Profile>
      <Service>
         <!--Service profile go here..... -->
      </Service>
      <DLs>
         <DL name="ARTE" jointTime="Jan. 23, 17:34:19 UTC 2006"/>
         <DL name="ImpECt" jointTime="Jan. 23, 17:35:34 UTC 2006"/>
         <!-- to be enriched -->
      </DLs>
      <DeploymentData>
         <ActivationTime value="Jan. 23, 17:33:45 UTC 2006"/>
         <TerminationTime value=""/>
         <SerializedState>a logical file name go here</SerializedState>
         <!-- to be enriched -->
      </DeploymentData>
      <AccessPoint>
         <ArchitecturalPattern>Factory</ArchitecturalPattern>
         <EndpointReference>
            <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
               <xs:import
schemaLocation="http://www.w3.org/2005/08/addressing/ws-addr.xsd"/>
            </xs:schema>
         </EndpointReference>
         <FactoryURI>
   http://localhost:8080/wsrf/services/factory/QueryParserFactoryService
         </FactoryURI>
      </AccessPoint>
      <DeployedPackagesList>
         <Package>
            <PackageName>name1</PackageName>
            <ConfigurationParameters>
               <Parameter name="" value=""/>
               <Parameter name="" value=""/>
            </ConfigurationParameters>
            <!-- to be enriched -->
         </Package>
      </DeployedPackagesList>
      <!-- an open RunningInstance-specific section go here...-->
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
         <xs:element name="SpecificData">
         <xs:complexType>
            <xs:sequence>
               <xs:any minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
         </xs:complexType>
         </xs:element>
      </xs:schema>
   </Profile>
</DILIGENTResource>
```

3.1.2.4 Collection profile

Collections are aggregation of information objects with explicit relationships with other information objects. The profile of a Collection includes the definition of the Collection. i.e. the set of criteria (e.g. membership conditions) that materialize the Collection by identifying

the information objects that populate the Collection itself. A Collection is *virtual* when there exists only its definition. A Collection is *materialized* when it is populated following its definition criteria.

These criteria are:

1) a set of object identifiers
2) the name of one or more Collections, e.g. the Collection is defined as an aggregation of other Collections
3) a predicate, e.g. a boolean expression that is used to populate a Collection.

Predicates may be (1) plain matches on the storage properties, (2) identical archive origin, or (3) complex metadata predicates (various operators) which must be evaluated by the Metadata Management (see Section 5.2.2).

Collection profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<DILIGENTResource>
   <UniqueID><!-- to be assigned by the RegistrationService--></UniqueID>
   <ResourceType>Collection</ResourceType>
   <AuthorizationPolicies/>
   <Profile>
   <Name>JDistributed IR: Jamie Callan papers</Name>
      <Description>This Collection contains the papers about information
retrieval, and in particular about distributed information retrieval
produced by Prof. Jamie Callan.
      </Description>
      <LastUpdate>Jan. 30, 17:53:29 UTC 2006</LastUpdate>
      <MembershipCondition>
      <!-- the language to specify membership condition in under
investigation -->
      </MembershipCondition>
   </Profile>
</DILIGENTResource>
```

3.1.2.5 gLiteResource profile

A gLiteResource profile includes information about *(i)* gLite services, *(ii)* Computing Elements, and *(iii)* Storage Elements of the underlying grid infrastructure. Its main section is an XML structure compliant with the XML Schema for GLUE Schema v1.2 with the exception of the *Host* element (not included in this profile because it is modeled with its own Model, see Section 3.1.2.7).

gliteResource profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<DILIGENTResource>
   <UniqueID><!-- to be assigned by the RegistrationService--></UniqueID>
   <ResourceType>gLiteResource</ResourceType>
   <AuthorizationPolicies/>
   <Profile>
      <!-- a section compliant with the GLUE Schema v1.2 go here.....-->
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:import
schemaLocation="http://infnforge.cnaf.infn.it/glueinfomodel/uploads/Spec/G
LUESchema12.xsd" namespace=""/>
      </xs:schema>
   </Profile>
</DILIGENTResource>
```

3.1.2.6 CS profile

The main section of the CS profile reports the CS specification created by the Process Management services (see Section 5.4), which are also in charge of registering the profile.

As described in [20], the CS specification should include information referring to, among others: list of services, execution costs, success probability, compensation, rollback and number of restart times (when the activity can be tried again).

Such a specification is expressed with a process specification language that is an enriched version of the Business Process Execution Language for Web Services[10] (BPEL4WS) and WSDL specifications.

CS profile example

```
<?xml version="1.0" encoding="UTF-8"?>
<DILIGENTResource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <UniqueID>
      <!-- to be assigned by the RegistrationService-->
   </UniqueID>
   <ResourceType>CS</ResourceType>
   <AuthorizationPolicies/>
   <Profile>
      <process name="createNewProcess"
targetNamespace="http://www.diligentproject.org/createNewServiceSpecificat
ion">
         <partners>
            <partner name="user" serviceLinkType="processDesignLink"
myRole="processDesignService"/>
            <partner name="processDesign" serviceLinkType="userLink"
myRole="newProcessRequester" partnerRole="newProcessService"/>
            <partner name="processResourceSystem"
serviceLinkType="processSaveLink" myRole="processSaveService"/>
         </partners>
         <containers>
            <container name="createNewCS" messageType="createnewCSMessage"/>
            <container name="exploreCSSpecification"
messageType="exploreCSSpecificationMessage"/>
            <container name="queryToDIS" messageType="queryToDISMessage"/>
            <container name="validateCS" messageType="validateCSMessage"/>
         </containers>
         <flow>
            <links>
               <link name="create-to-ProcessDesign"/>
               <link name="explore-to-ProcessResourceSystem"/>
               <link name="query-to-DIS"/>
               <link name="validate-to-ProcessDesign"/>
            </links>
            <receive partner="CSDesignPortlet" portType="createNewCSPT"
operation="createNewCSOP" container="createNewCS">
               <source linkName="create-to-ProcessDesign"/>
            </receive>
            <invoke partner="processDesign"
portType="exploreCSSpecificationPT" operation="exploreCSSpecificationOP"
inputContainer="exploreCSSpecification">
               <target linkName="create-to-ProcessDesign"/>
               <source linkName="explore-to-ProcessResourceSystem"/>
            </invoke>
            <invoke partner="processResourceSystem" portType="queryToDISPT"
operation="queryToDISOP" inputContainer="queryToDIS">
               <target linkName="explore-to-ProcessResourceSystem"/>
               <source linkName="query-to-DIS"/>
            </invoke>
            <receive partner="processDesign" portType="validateCSPT"
operation="validateCSOP" container="validateCS">
               <target linkName="validate-to-ProcessDesign"/>
            </receive>
         </flow>
      </process>
   </Profile>
</DILIGENTResource>
```

[10] BPEL4WS - http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

3.1.2.7 DHN profile

The DHN profile includes:

- the hardware and third party software characteristics;
- the location;
- the network configuration;
- the site (for proximity requirements);
- any other information compliant with the *Host* GLUE Schema v1.2;
- information related to the deployed packages.

The main section of the DHN profile is an XML structure compliant with *Host* element defined in the XML Schema for GLUE Schema v1.2 [22].

DHN profile Example

```
<?xml version="1.0" encoding="UTF-8"?>
<DILIGENTResource>
   <UniqueID><!-- to be assigned by the RegistrationService--></UniqueID>
   <ResourceType>DHN</ResourceType>
   <AuthorizationPolicies/>
   <Profile>
      <DHNDescription>
      <!-- a section compliant with the Host element of the GLUE Schema
v1.2 go here...-->
         <xs:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xs:import
schemaLocation="http://infnforge.cnaf.infn.it/glueinfomodel/uploads/Spec/G
LUESchema12.xsd" namespace=""/>
         </xs:schema>
      </DHNDescription>
      <Site>
         <!-- a section with related site information go here...-->
      </Site>
      <DeployedPackagesList>
         <Package>
            <PackageName>name1</PackageName>
            <ConfigurationParameters>
               <Parameter name="" value=""/>
               <Parameter name="" value=""/>
            </ConfigurationParameters>
            <DLs>
               <DL name="ARTE 1"/>
               <DL name="ImpECt 3"/>
            </DLs>
            <!-- to be enriched -->
         </Package>
         <Package>
            <PackageName> name2</PackageName>
            <ConfigurationParameters>
               <Parameter name="" value=""/>
               <Parameter name="" value=""/>
            </ConfigurationParameters>
            <DLs>
               <DL name="ARTE 2"/>
            </DLs>
            <!-- to be enriched -->
         </Package>
      </DeployedPackagesList>
   </Profile>
</DILIGENTResource>
```

3.2 Users and Roles

To represent users capabilities, i.e. the way a user interacts with the system, DILIGENT introduces used roles. The roles we have modeled are the same identified during the functional specification phase and reported in D1.1.1 [1].

A role grants the actor some specific rights (those assigned by resource owners to that role). Roles are hierarchically related by a specialization relationship that makes the specialized role to have all the rights of the parent and a number of other specific ones. We reproduce this hierarchy in Figure 10.

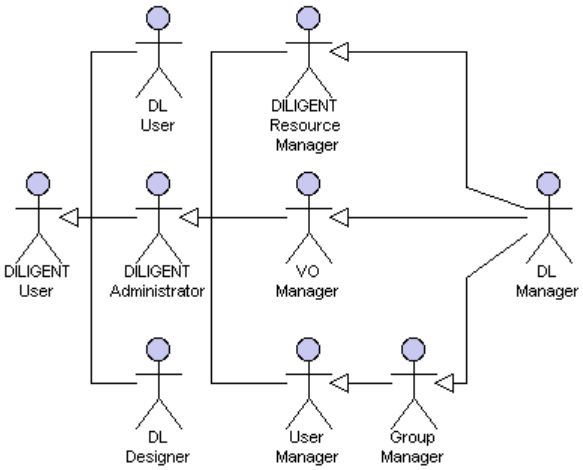


Figure 10: DILIGENT actors

For the purpose of this document, suffice it to know that:

- a *DL Designer* can design a VDL;
- a *DL Manager* can approve/modify the VDL design and start the DL creation process;
- a *VO Manager* is entitled to manage a Virtual Organization;
- a *Resource Manager* is allowed to register new DILIGENT Resources;
- a *DL User* is a generic end-users that access to the DL functionality;
- a *User/Group Manager* can register and manage users/groups of users.

D1.1.1 provides an extensive description of each role together with the functionality it can access.

3.3 Virtual Organizations

The landmark paper *Anatomy of the Grid* [7] defines Virtual Organizations (VOs) as dynamic pools of heterogeneous resources. The access to and the usage of such resources within a VO is ruled though a set of sharing rules which applies to the whole VO regardless of its members. Within the VO, access to resources can further restricted for any user enabling finer-grained authorization. Thus an authorization is a 4-tuple function of:

Resource, SharingRule, Action, User/Role

As a consequence of what we have described so far, Virtual Organizations in DILIGENT are dynamic aggregations of DILIGENT Resources. The VO Data Model reported in Figure 11 explains how Virtual Organizations are structured in DILIGENT.

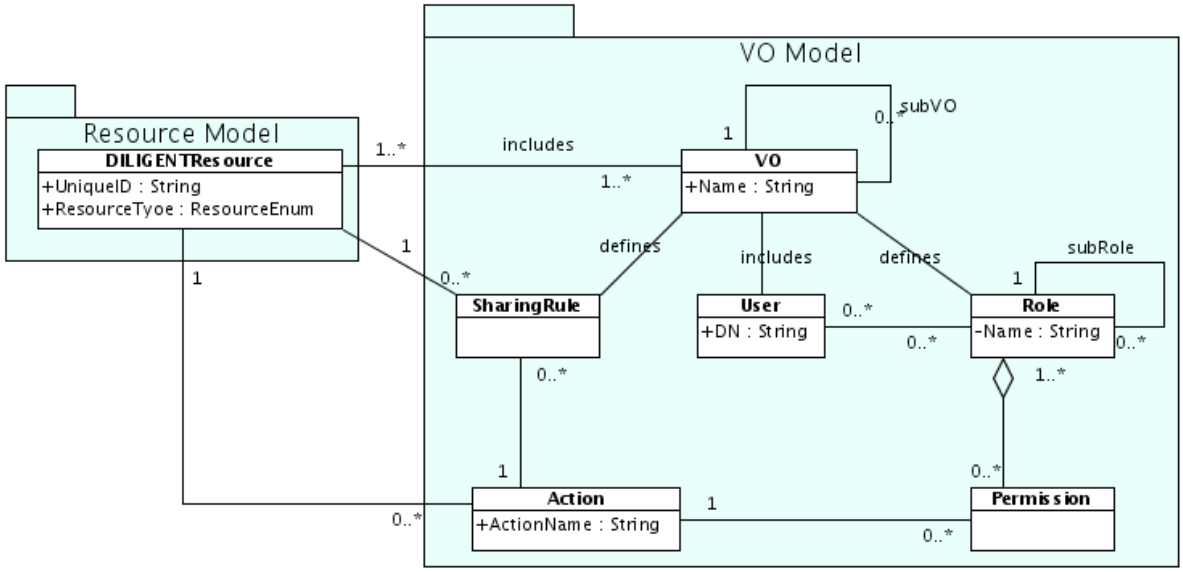


Figure 11. VO Model

A *VO* includes DILIGENT users and resources. A *User* is identified through its Distinguished Name (DN) and a *DILIGENTResource* is identified through its UniqueID assigned at the registration time. A very important thing to be pointed out is that Virtual Organizations do not define users and resources. Each VO only includes references to existing users and resources, thus they can be part of multiple Virtual Organizations at the same time. *Roles*, *Sharing Rules* and *Permissions*, on the contrary, are defined in VOs and inherited by sub-VOs.

3.3.1 VOs Hierarchy

In DILIGENT there is a three-level hierarchy of VOs – although the above defined VO model does not impose that:

1. at the top level, there exists a global VO named *DILIGENT VO* ;
2. such a VO is partitioned into other VOs, one for each end-user community (for instance, the ARTE VO, the ImpECT VO, etc.);
3. finally, at the bottom of this hierarchy, there are the sub-VOs that support the mapping of the VDL definitions into a real DLs.

At the registration time, a new DILIGENT Resource is automatically registered in the DILIGENT VO. Furthermore, the Resource Manager (i.e. the owner of the resource) can specify which user community(-ies) can access the resource and with which rights by fixing the appropriate sharing rules. When the Creation and Management services (see Section 0) set up a new DL, they add the resource to the DL's sub-VO if the resource is part of the rising DL.

3.3.2 Secure Communication in DILIGENT

In this section we address the Security Model adopted to have secure communication over the network. It is placed in this part of the document since it is the mechanism that regulates under which conditions a service client can access RunningInstances and DHN resources. The Model is based on the security framework provided by Java WS Core. Figure 12 shows the Security Model that DILIGENT adopts to address secure communications. The aim of the diagram is to show which security options are available to DILIGENT Services and Clients through the security descriptor approach.

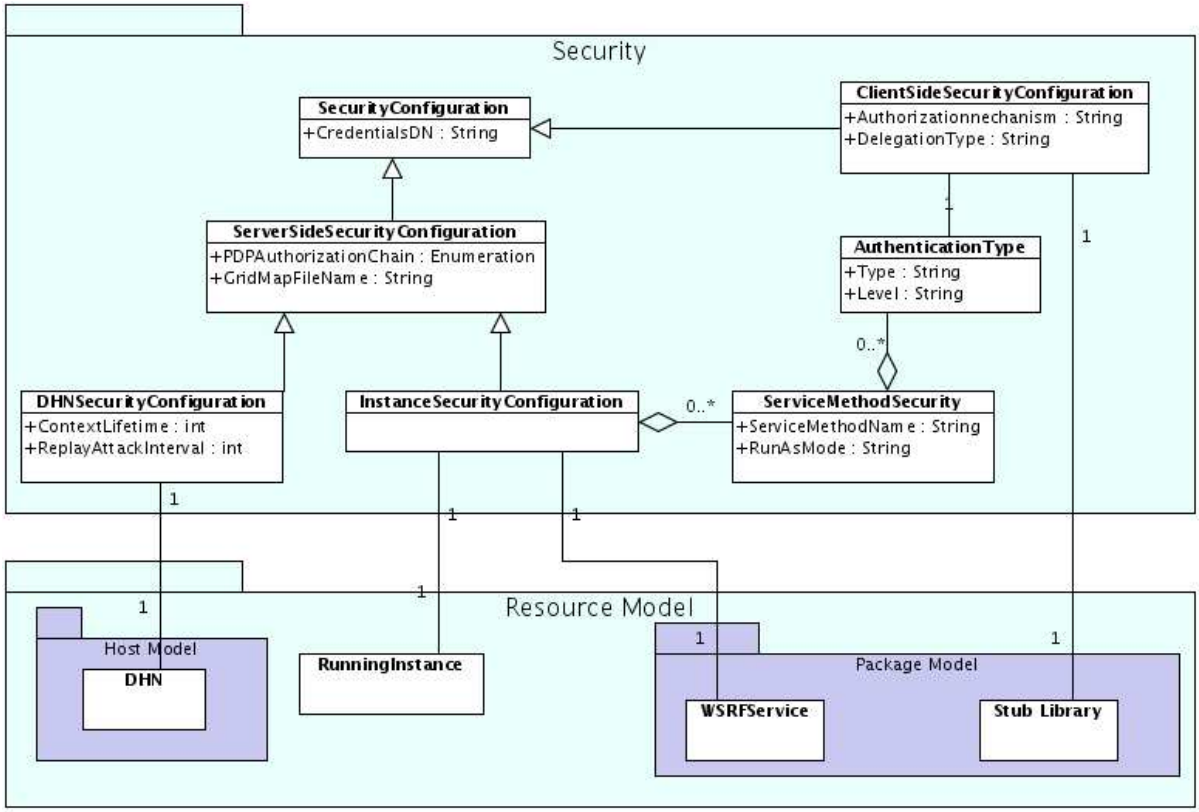


Figure 12. Security communication

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

Each method of the service can use different, and multiple, authentication mechanisms (MessageSecurity, WSSecureConversation, TSL). Moreover, the descriptor contains a list of Authorization handlers to be used during enforcement of authorization policies. Finally, DHN and Clients (Stub Library) can also define their own Security Descriptors.

For a detailed description of the content of the Service Security Descriptor and how to create and use Security Descriptors, see the Java WS Core documentation[11].

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

4 MIDDLEWARE INTEGRATION

The DILIGENT system is built by integrating grid and digital library technologies. The characteristics of the grid framework making it particularly appealing to the goal of the DILIGENT project reside in the Ian Foster et Al.'s ideas and works introducing the grid technology, in particular the conception of "*coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations*. The sharing that we are concerned with is not primarily file exchange rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization (VO)" [7].

However, this idea is far from being concretely and fully implemented. There are various initiatives and project that aim at making this "dream" as "real" as possible. In Europe, the EGEE project has been funded with the aim to provide researchers in academia and industry with access to major computing resources, independent of their geographic location. Two objectives of this project are relevant with respect to the DILIGENT objectives:

- to build a consistent, robust and secure Grid network that will attract additional computing resources;
- to continuously improve and maintain the middleware in order to deliver a reliable service to users.

The DILIGENT project decided to build over the Grid infrastructure provided by the EGEE project because: (i) they release a service grid infrastructure which is available to scientists 24 hours-a-day; (ii) the infrastructure is built on the EU Research Network GÉANT and exploit Grid expertise generated by many EU, national and international Grid projects to date; (iii) the infrastructure is providing a production quality grid infrastructure spanning more than 30 countries with over 150 sites to a myriad of other applications from various scientific domains, including Earth Sciences, High Energy Physics, Bioinformatics and Astrophysics; (iv) the infrastructure is empowered by an innovative, robust and reliable middleware, named *gLite*.

As a consequence the DILIGENT architecture has been designed to communicate with and rely on the facilities provided by the gLite middleware that covers most of the needs required by the DILIGENT system.

4.1 gLite middleware architecture

In this section we present the Grid middleware released by the EGEE project – named gLite. It is influenced by the requirements of Grid applications, the ongoing work in the Global Grid Forum (GGF) on the Open Grid Services Architecture (OGSA), and the previous experience from other Grid projects such as EU DataGrid (EDG), LHC Computing Grid (LCG), AliEn, Virtual Data Toolkit (VDT). It includes among others Globus, Condor and NorduGrid.

The gLite Grid services follow a Service Oriented Architecture (SOA) which facilitates interoperability among Grid services and easier compliance with upcoming standards, such as OGSA, that are also based on these principles.

Figure 13 shows gLite high level services, which can thematically be grouped into 5 service groups plus associated APIs and CLIs. All gLite services are briefly described in the following subsections. More detailed information can be found in the middleware webpage http://glite.web.cern.ch/glite/ and in the gLite development team (EGEE JRA1) webpage http://egee-jra1.web.cern.ch/egee-jra1/.

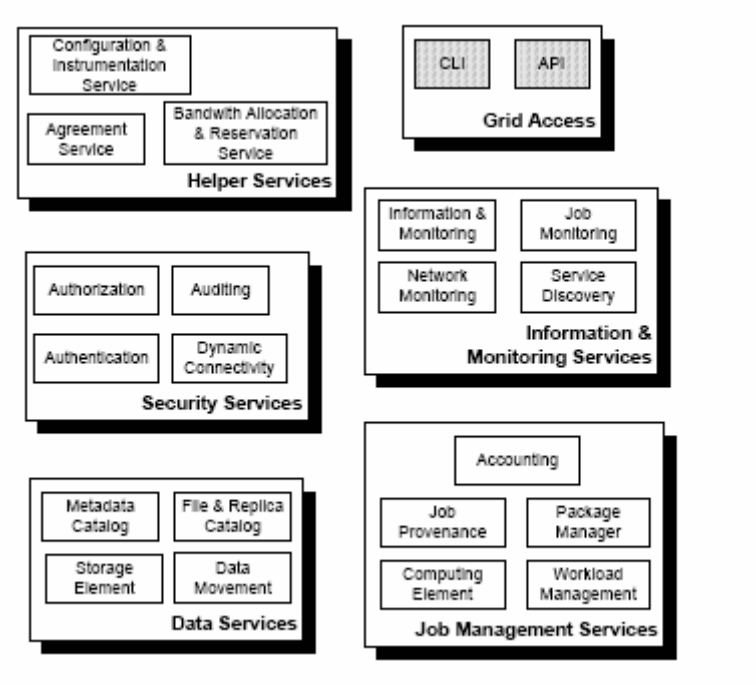


Figure 13. glite middleware architecture

4.1.1 Security Services

Includes the Authentication, Authorization, and Auditing services which enable the identification of entities (users, systems, and services), allow or deny access to services and resources, and provide information for post-mortem analysis of security related events. It also provides functionality for data confidentiality and a dynamic connectivity service.

4.1.1.1 Virtual Organization Membership Service

Virtual Organization Membership Service (VOMS) serves as a central repository for user authorization information, providing support for sorting users into a general group hierarchy, keeping track of their roles, etc. Its functionality may be compared to that of a Kerberos KDC server. The VOMS Admin service is a web application providing tools for administering member databases for VOMS.

4.1.1.2 gLite Security Utilities

The gLite Security Utilities module contains the CA Certificates distributed by the EU Grid PMA. In addition, it contains a number of utility scripts needed to create or update the local grid mapfile from a VOMS server and periodically update the CA Certificate Revocation Lists.

4.1.2 Information and Monitoring Services

The information system is used to store and publish information about the different parts of a gLite based grid infrastructure (services, sites etc.) and to query this information by interested users and services via the service discovery. These services rely on registering the location of publishers of information and what subset of the total information they are publishing. This allows consumers to issue queries to the information system while not having to know where the information was published.

4.1.2.1 Relational Grid Monitoring Architecture

The Relational Grid Monitoring Architecture (R-GMA) is the Information and Monitoring Service of gLite. It is based on the Grid Monitoring Architecture (GMA) from the Grid Global Forum (GGF), which is a simple Consumer-Producer model that models the information

infrastructure of a Grid as a set of *consumers* (that request information), *producers* (that provide information) and a central *registry* which mediates the communication between producers and consumers. R-GMA offers a global view of the information as if each Virtual Organisation had one large relational database.

The functionality of the R-GMA system can be logically split in a server part and several clients. The R-GMA server is the server part of the R-GMA infrastructure that is used by the different producers and consumers and is divided into four components:

- R-GMA Server - The server component of the information system
- R-GMA Schema Server - Defines the schema in the information system
- R-GMA Registry Server - The registry server for the grid
- R-GMA Browser - Browses the information of the information system

The client part of R-GMA contains the producer and consumers of information. There is one generic client and four specialized clients to deal with certain types of information:

- Generic Client - A generic set of APIs for different languages and CLIs.
- R-GMA Service Publisher - Client to publish the existence and status of services.
- R-GMA Site Publisher - Client to publish the existence of a site.
- R-GMA GIN - Client to extract information from MDS and to republish it to R-GMA.
- R-GMA data archiver - Client to make the data that is coming from the R-GMA site-publisher, service-publisher and GIN constantly available.

4.1.2.2 Service Discovery

The Service Discovery module is the counterpart to the information system. It allows the different gLite modules to discover the endpoint of other gLite modules they are interested in. The Service Discovery module can use several information systems

- R-GMA
- Berkely Database Information Index (BDII)
- Files that contain the necessary information

4.1.3 Job Management Services

The main services related to job management/execution are the computing element, the workload management, accounting, job provenance, and package manager services.

The Workload Management System (WMS) comprises a set of grid middleware components responsible for the distribution and management of tasks across grid resources, in such a way that applications are conveniently, efficiently and effectively executed.

4.1.3.1 Workload Manager

The core component of the Workload Management System is the Workload Manager (WM), whose purpose is to accept and satisfy requests for job management coming from its clients.

For a computation job there are two main types of request: submission and cancellation. In particular the meaning of the submission request is to pass the responsibility of the job to the WM. The WM will then pass the job to an appropriate Computing Element for execution, taking into account the requirements and the preferences expressed in the job description. The decision of which resource should be used is the outcome of a matchmaking process between submission requests and available resources.

4.1.3.2 Logging and Bookkeeping

The Logging and Bookkeeping service (LB) tracks jobs in terms of events (important points of job life, e.g. submission, finding a matching CE, starting execution etc.) gathered from various WMS components as well as CEs.

Each event type carries its specific attributes. The entire architecture is specialized for this purpose and is job-centric. The events are gathered from various WMS components by the LB producer library, and passed on to the locallogger daemon. The locallogger's task is storing the accepted event in a local disk file. Further on, event delivery is managed by the interlogger daemon. It takes the events from the locallogger, and repeatedly tries to deliver them to the destination bookkeeping server (known from the JobId) until it succeeds finally.

Besides querying for the job state actively, the user may also register for receiving notifications on particular job state changes (e.g. when a job terminates).

4.1.3.3 Computing Element

The Computing Element (CE) is the service representing a computing resource. Its main functionality is job management. The CE may be used by a generic client: an end-user interacting directly with the Computing Element, or the Workload Manager, which submits a given job to an appropriate CE found by a matchmaking process.

For job submission, the CE can work in push model (where the job is pushed to a CE for its execution) or pull model (where the CE asks the WMS for jobs).

Besides job management capabilities, a CE must also provide information describing itself. In the push model this information is published in the information Service, and it is used by the match making engine which matches available resources to queued jobs. In the pull model the CE information is embedded in a "CE availability'" message, which is sent by the CE to a Workload Management Service. The matchmaker then uses this information to find a suitable job for the CE.

4.1.3.4 Torque Resource Manager

Torque Resource Manager (Tera-scale Open-source Resource and QUEue manager) is a resource manager providing control over batch jobs and distributed computing nodes. The torque system is composed by a pbs_server which provides the basic batch services such as receiving/creating a batch job or protecting the job against system crashes. The pbs_mom places the job into execution when it receives a copy of the job from a Server. The mom_server creates a new session as identical to a user login session as possible. It also has the responsibility for returning the job's output to the user when directed to do so by the pbs_server. The job scheduler is another daemon which contains the site's policy controlling which job is run and where and when it is run. The scheduler appears as a batch Manager to the server. The scheduler being used by the torque module is *MAUI*.

4.1.3.5 Worker Node

The gLite Standard Worker Node is a set of clients required to run jobs sent by the gLite Computing Element via the Local Resource Management System. It currently includes the gLite I/O Client, the Logging and Bookeeping Client, the R-GMA Client and the WMS Checkpointing library. The gLite Torque Client module can be installed together with the WN module if Torque is used as a batch system.

4.1.3.6 DataGrid Accounting System

The DataGrid Accounting System (DGAS) software aims at being a full featured distributed Grid accounting toolkit. Since it is conceived and designed to be completely grid oriented, it is fully distributed without having a central repository of accounting information. It instead

relies upon a network of independent accounting servers used to keep the accounting/transaction records of groups of GridUsers and GridResources.

DGAS can be used to account classic Computational Usage Records like CPU Time, memory usage and so on. It can also be used as an Economic Accounting system, treating information about the cost of the jobs executed by each GridUser on the single GridResources.

4.1.4 Data Services

The three main service groups that relate to data and file access are: storage element, catalog services, and data movement services. In all of the data management services described below the granularity of the data is on the file level. However, the services are generic enough to be extended to other levels of granularity. Data sets or collections are a very common extension where the information about which files belong to a dataset may be kept in an application metadata catalog.

To the user of the EGEE data services the abstraction that is being presented is that of a global file system. The access to the files is controlled by Access Control Lists (ACL).

4.1.4.1 gLite I/O

GLite I/O server consists basically of the server of the AliEn Aiod project, modified to support GSI authentication, authorization and name resolution plug-ins, together with other small features and bug fixes. GLite I/O allows access to remote files using the dcap or the rfio client library.

It can interact with the FiReMan Catalog, the Replica Metadata Catalog and Replica Location Service, with the File and Replica Catalogs or with the Alien File Catalog.

4.1.4.2 Fireman Data Catalog

On the Grid, the user identifies files using Logical File Names (LFN). The LFN is the key by which the users refer to their data. Each file may have several replicas, i.e. managed copies. The management in this case is the responsibility of the File and Replica Catalog.

The replicas are identified by Site URLs (SURLs). Each replica has its own SURL, specifying implicitly which Storage Element needs to be contacted to extract the data. The SURL is a valid URL that can be used as an argument in an SRM request. Usually, users are not directly exposed to SURLs, but only to the logical namespace defined by LFNs. The Grid Catalogs provide mappings needed for the services to actually locate the files. To the user the illusion of a single file system is given.

4.1.4.3 Metadata Catalog

Metadata is in general a notion of "data about data". There are many aspects of metadata, like descriptive metadata, provenance metadata, historical metadata, security metadata, etc. Whatever is its nature, metadata is associated with items, named to be unique within the catalog. The gLite Metadata Catalog makes no assumption on what each of these items represents. To each of these items a user may associate two sets of information:

- Groups of key/value pairs (attributes), defined within schemas;
- Permissions, just like in the gLite Fireman catalog, expressed via BasicPermissions and ACLs.

The functionality offered allows the user to manage the schemas, set and get values of attributes, perform queries using metadata values and manage the access permissions on each individual item.

At time of writing this report, this component is not supported any more and it is replaced by the AGMA Metadata catalog. The added features provided by this new component appear very valuable because they include, but are not limited to: (i) a streaming and SOAP

front-end; (ii) an interactive client for the streaming front-end; (iii) client APIs for the streaming client (C++, Java and Python).

4.1.4.4 File Transfer Service

The data movement services of gLite are responsible to securely transfer files between Grid sites. The transfer is always performed between two Storage Elements having the same transfer protocol available to them (usually gsiftp). The gLite File Transfer Service is composed of the File Transfer Service web service (responsible for managing data transfers and placements), and a number of file transfer agents.

The File Transfer Service is responsible for the actual transfer of the file between the SEs. It takes the source and destination names as arguments and performs the transfer. The FTS is managed by the site administrator, i.e. there usually is only one such service serving all VOs.

4.1.4.5 File Transfer Agents

The File Transfer Agents perform Data Transfer and Placement actions. We distinguish two different kinds of agent: the Channel Agent and the VO Agent. The Channel Agent is responsible for managing all the file transfers through a channel, i.e. the entity that represents the physical, monodirectional link between two sites: this agent will fetch the file transfer requests from a Queue and submit them to the configured File Transfer Service. The other type of agent, the VO Agent, is in charge of performing all the actions that are related to a specific Virtual Organization, which involves applying VO policies, managing catalog interactions and running VO custom actions. Moreover, we distinguish between two possible VO Agent deployment types:

- File Transfer Service (FTS) Agent: This agent manages jobs where the source and destination contain Physical File Names (SURLs or TURLs). No catalog interaction is required
- File Placement Service (FPS) Agent: Extends the previous model adding the interaction with a Catalog Service, in order to retrieve the source and destination physical file names from the Logical File Names (LFNs and GUIDs) and source and destination sites. Once a transfer is completed, the new replicas are registered into the appropriate catalog.

4.1.5 Helper Services

In addition to the services described above, a Grid infrastructure may provide a set of helper services that aim at providing a higher level abstraction (like for instance Workflow services or market mechanisms), better quality of service (like for instance reservation and allocation services), or better manageability of the infrastructure (like for instance configuration services). In the gLite current architecture we include three examples of such helper services, namely the Configuration and Instrumentation Service, the Bandwidth Allocation and Reservation Service, and the Agreement Service.

The Configuration and Instrumentation Service provides a common, standard-based configuration and instrumentation functionality to the gLite services. In particular, the configuration part is concerned with dynamic changes in the configuration of a Grid service while the instrumentation part is concerned with obtaining the state of a Grid service.

The Bandwidth Allocation and Reservation Service provides mechanisms to control and balance the usage of the network and to categorize and prioritize traffic flows so that users and the layers of Grid middleware receive the required level of service from the network. Finally, the Agreement Service allows the dynamic establishment of Service Level Agreements between agreement initiators (the service requestors) and the service providers.

However, at time of writing of this report, the helper Services components are not yet available at production quality level. For the sake of completeness we reported the designed components and their main functionality.

4.1.6 Grid Access

Most of the gLite service APIs and CLIs are provided in a common deployment package called User Interface (UI). The gLite user Interface is a suite of clients and APIs that users and applications can use to access the gLite services. The gLite User Interface includes the following components:

- Data Catalog command-line clients and APIs;
- Data Transfer command-line clients and APIs;
- gLite I/O Client and APIs;
- R-GMA Client and APIs;
- VOMS command-line tools;
- Workload Management System clients and APIs;
- Logging and bookkeeping clients and APIs.

4.2 gLite use and exploitation

The integration of two technologies can be performed and influence the design of the system at various levels. Moreover, the integration can have different goals. In particular, within the DILIGENT project two of the objectives are related to grid exploitation, i.e. (*i*) *to open up the grid technology to a brother range of research and industrial communities*, and (*ii*) *to promote the cross-fertilization between DL and Grid technology domains that will foster synergies and advances in both research areas*.

The first exploitation of the grid technology is the usage of the gLite middleware as the enabling technology for developing the DILIGENT infrastructure. This infrastructure on the one hand enables DILIGENT services to exploit the Grid capabilities and resources also provided by third-parties organizations; on the other hand provides the DILIGENT based end-user applications with a transparent and easily access to a consistent, robust and secure Grid network providing unexpected storage and computational capabilities.

The second level of exploitation of the gLite middleware is tailored to realize the DILIGENT services that are based on complex and time consuming algorithms with a reasonable performance, and to enhance the DL storage and preservation techniques basing them on the mechanisms for data replication and security handling developed in the Grid area. These are two examples of such kind of exploitation:

- The CS Management Service, i.e. the DILIGENT set of services dedicated to the design and implementation of process (also known as workflows, or compound services), has been enriched with the capability to use jobs within a process. A process is a "program" that combines services in a novel way to provide new functionality. Thanks to this characteristic, jobs that will run on the gLite-based infrastructure can seamlessly be included in a process and contribute to the provision of DILIGENT high-level functionalities. For instance, it becomes feasible to define a process that combines a number of jobs in a set of dependent and independent chain of tasks. The execution of this process is then managed and monitored by the CS Management Service but it exploits the available (at execution time) computational Grid resources and it is regulated by the authentication and authorization framework enforced by the Grid infrastructure.
- The Data Management part of DILIGENT relies on the gLite data management capability to perform physical storage of unstructured binary content. Binary content (like images, PDF documents, scientific data, etc.) are stored as files in gLite storage

elements even if they still have a placeholder representation in the DILIGENT storage model and may, thus, be provided with an arbitrary number of storage properties and take part in any inter-object relationship. In detail, DILIGENT Data Management services rely on (*i*) the FiReMan (file & replica) catalog, (*ii*) a gLite I/O server and client installation, and (*iii*) an appropriate storage backend (dCache, DPM, Castor, etc.). Moreover, the DILIGENT Data Management services use the File Transfer Service (FTS) to transport binary content (materialized in files) across the Grid. For instance, upon file storage requests, the file to be stored will be (*i*) picked up from a specified location in the local file system of the invoking client service, (*ii*) transported to the respective data management node, and (*iii*) placed into a specific directory for incoming files. Vice versa, content access requests (*i*) fetch content from gLite and place it into a specific directory on the data management site, (*ii*) transport that file to the invoking client service site, and (*iii*) place it into a specified directory.

Finally, a bridge between the two environments has been built to support this type of interactions. This bridge is needed for two reasons. On the one hand, it is required to achieve a common implementation of the controlled access to and sharing of resources because the two environments are based on different flavors of technologies. On the other hand, the heterogeneity of the two environments needs a mediator to cross over the administrative boundaries. For instance, each party must be able to validate the identities issued by the other parties; each party must be free to assign rights to identities in an independent manner. This bridge is spread over:

- the DILIGENT Information Service that harvests all the information maintained by one or more gLite based information services and exposes them to all the other DILIGENT services. This allows simplifying the discovery and monitoring of resources even if multiple gLite based Grid infrastructures are joined.
- the DILIGENT Virtual Organization Support Service that provides the authentication and authorization framework to allow a DILIGENT service to obtain valid credential for accessing the underlying Grid resources.

5 THE DILIGENT ARCHITECTURE

This section presents the architecture of the DILIGENT system starting from the outcomes of the service specification reports released by project partners. The architecture is presented by grouping the services following their functional area:

- Digital Library (DL) Creation & Management
- Index & Search
- Content & Metadata Management
- Process Management
- User-Community Specific Applications

For each area, an integrated vision is reported. Then, each service belonging to that area is presented in its real components. A *component* of a service is "a part" of that service that can be hosted in different networked locations. Following the terminology introduced in Section 3, these locations are named *DHNs* (DILIGENT Hosting Nodes). To become a DHN, a node must have installed the following software:

- a Java WS Core hosting environment where DILIGENT services can be deployed;
- a minimal DILIGENT software (a component of the Keeper, see section 5.1.3) that collaborates in the local deployment of the components

In the integrated diagrams presented for each area, all the "boundary" classes of the logical views of the single services (reported in the service specification reports [17] [18] [19] [20] [21]) are included; these classes become the public interface of the service components in the implementation phase. In these UML diagrams, the subsystems (light grey boxes) represent the components of each service. Therefore, they are the WSRF services, portlets, libraries and grid jobs that make concrete the system.

The main purpose here is to gain a general understanding of how the system is decomposed, and how the individual parts work together to provide the expected functionality by focusing on the "connectors" (see Section 2.1.1).

Software deployment

As stated, the deployable software released by DILIGENT is organized in components. The final goal of such a organization is their automatic and dynamic deployment on the appropriate DHN taking care of the requirements of a component and the characteristics of the available DHNs.

In order to reach such a goal, all the components must be physically packaged following the Package Model defined in Section 3.1.1.3. A *software package* compliant with this model includes the files to be installed as well as the rules describing software/packages dependencies and deployment instructions and it is candidate to be managed correctly by the DL Creation & Management services.

The compliance to the Model mainly means that a package must contain all the information requested by the Model for its particular package type (WSRFService, Portlet, Library or GridJob). Of course, a further agreement among the different services involved in the various management phases of the package must be achieved during the implementation of such services on the physical formats of the package types.

5.1 DL Creation & Management services

Under the name *DL Creation & Management* (DCM) there are grouped the services that bring together all the resources distributed across a Grid infrastructure and support the creation of new Digital Libraries.

The capabilities provided by the DCM services enable a virtual research organization to dynamically create and modify its own DLs by specifying a set of definition criteria that fits its needs and includes a number of requirements on the information space (e.g. publishing institutions, subject of the content, document type, level of replication) and on the services (e.g. service type, configuration, lifetime, availability, response time, others).

In particular, the DCM services manage the entire process that, ranging from the registration of the software packages to their remote deployment, allows creating, configuring, monitoring, maintaining, and disposing a DL. This process includes:

1. the creation of a trusted environment that ensures a controlled sharing of these resources by exploiting the VO mechanisms provided by the gLite middleware and the development framework (*Dynamic VO Support Service*)
2. the implementation of a global strategy offering a valuable use of the resources supplied by the DILIGENT infrastructure (*Broker & Matchmaker Service*)
3. the selection and automatic configuration of the best pool of resource to form a DL fulfilling the particular (and possibly temporal) needs of a end-user community (*VDL Generator Service*)
4. the orchestration needed to maintain up and running the pool of resources that populate the various DLs and to ensure measurable levels of fault tolerance and QoS (*Keeper Service*)

All the performed activities are supported by the *DILIGENT Information Service* that delivers a complete framework to monitor and discover resource information.

The DCM features rely on the respect of the constraints imposed by the various models presented in Section 3. The conformity to these models is the foundation for a correct management of a resource in the following cases:

- the resource registration
- the selection for the inclusion in a new DL
- the deployment phase
- the management of the service's life-cycle
- the information management
- the authentication and authorization process

Most of these operations are described in the following while a more detailed explanation of them is reported in [17].

Figure 14 depicts the overall and integrated picture of the DCM services and their relationships.

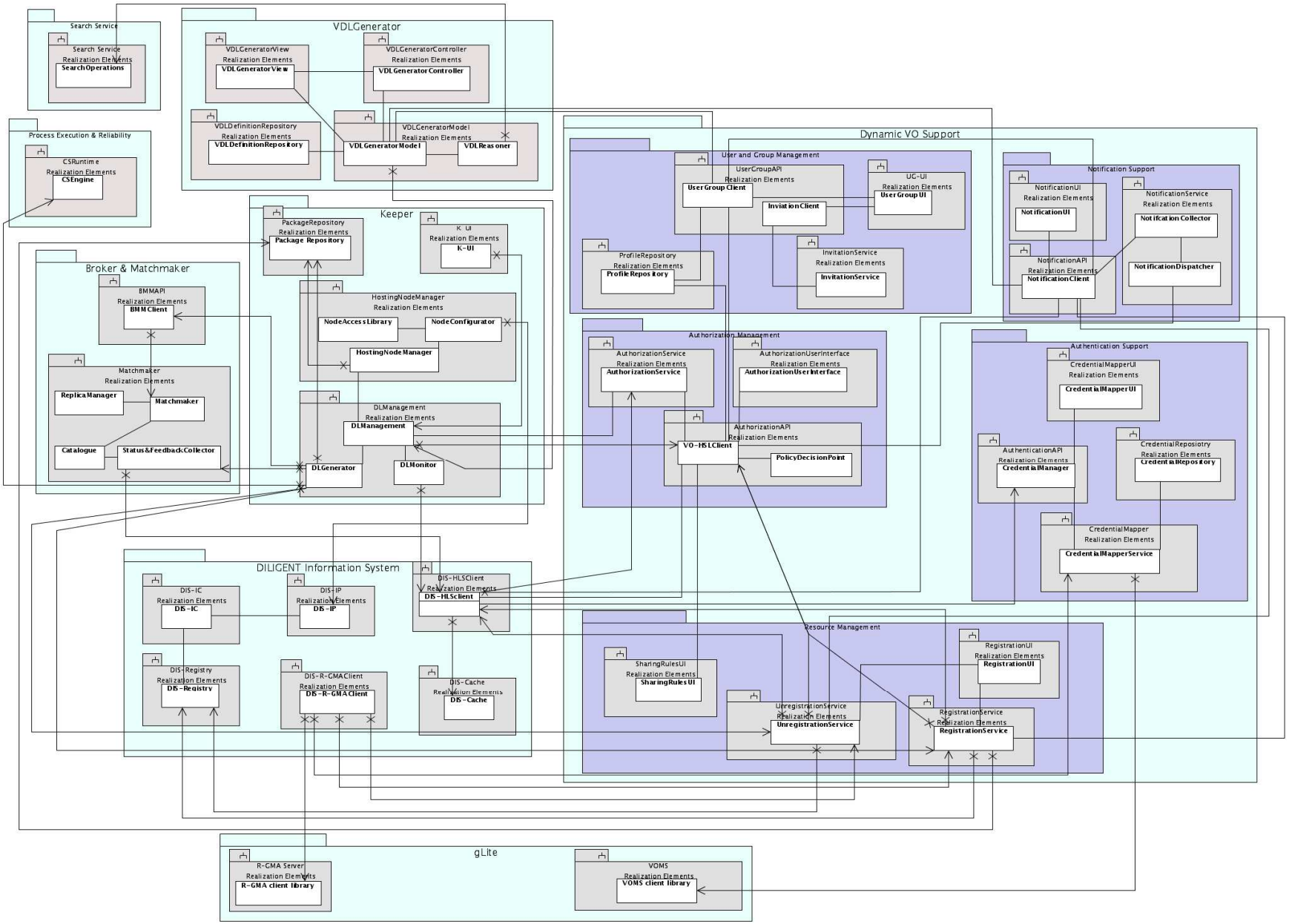


Figure 14. DL Creation & Management services – architectural View

As stated, each component is packaged in a *software package* that can be dynamically deployed on a DHN when needed and without any human interaction. The dynamic deployment of a package is a complex process that involves all the DCM services. The final result of this process is that a "piece of software" is moved to the appropriate DHN. A special case is the deployment of packages that "wraps" a WSRF service or a portlet. Once a package of this kind is deployed and activated, it becomes a RunningInstance. In Figure 15 we present the expected relationships between a generic RunningInstance and the DL Content & Management services. A Running Instance should:

- query the DIS-HLSClient to discover information about resources;
- interact with the AuthorizationAPI to manage its authorization policies;
- interact with the CredentialManager to verify credentials and identities;
- interact with the CredentialMapper to obtain a valid identity for the underlying gLite infrastructure;
- invoke the NotificationClient to notify users/groups (e.g. VO Managers, a particular user, etc.) of relevant changes in the status of the service;
- interact with the DIS-IP to publish the information related to the resources that it creates;
- interact with the Hosting Node Manager to update its profile.

In addition, a RunningInstance is periodically monitored by the DL Monitor component of the Keeper to verify if it is working properly.

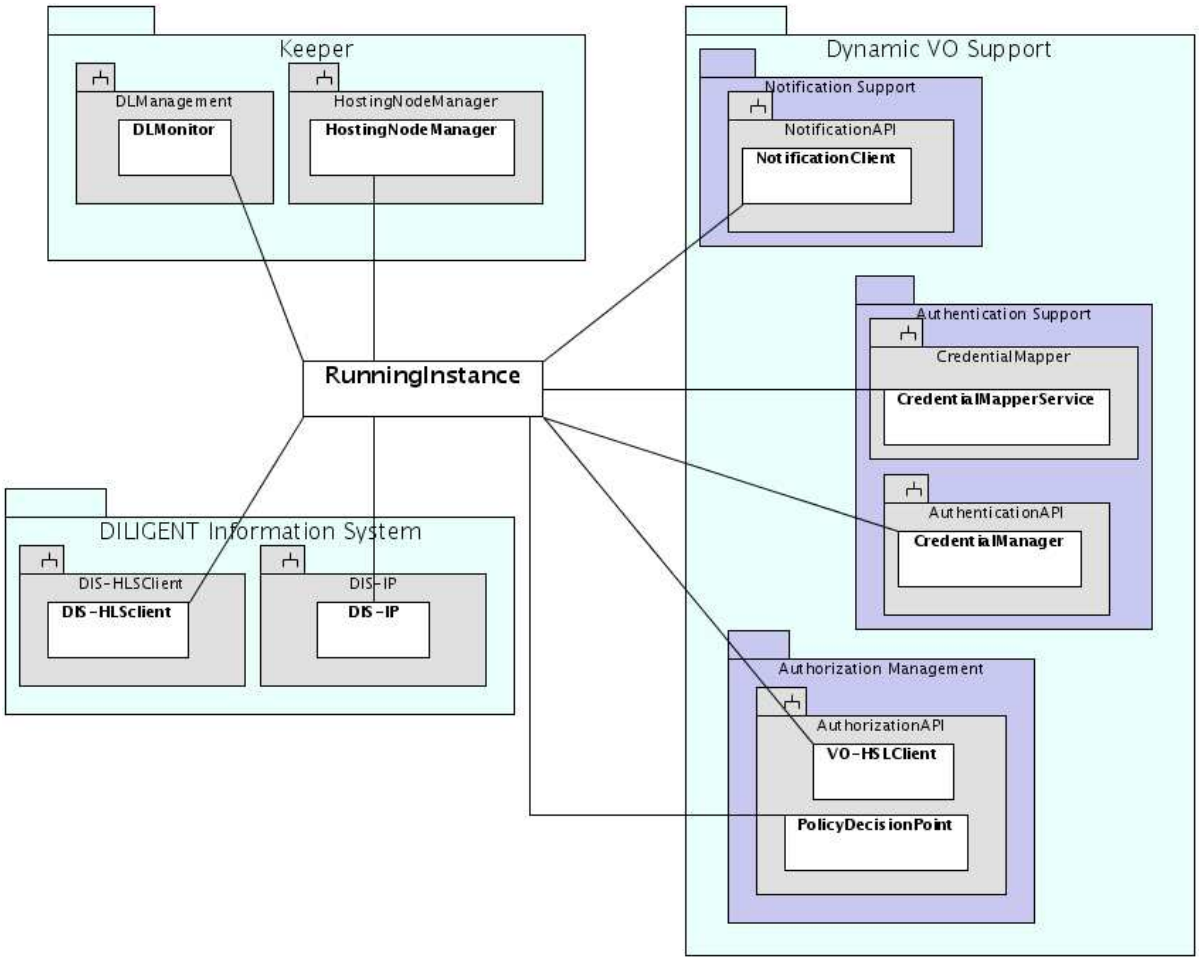


Figure 15. Generic RunningInstance interactions

Each of the following subsections presents a DCM service reporting a brief description and its components. Hereafter, we also highlight the most important interactions that emerge from Figure 14. Details about the DCM services are instead reported in the "D1.2.2 DL Creation & Management services specification" report [17].

5.1.1 DILIGENT Information Service

The DILIGENT Information Service (DIS) maintains the most updated information about the set of available DILIGENT resources (see Section 3.1) that compose the DILIGENT VO. This information includes:

- the profile of the resources
- the status of the DILIGENT running instances and DHNs

Once these data are collected, the DIS provides mechanisms for

- allowing users and services to discover which resources are part of a Virtual Organization (VO);
- monitoring those resources;
- querying arbitrarily structured resource information;
- initiating an action when an event occurs (triggering) starting from pre-configured conditions;
- archiving information to allow historical query execution.

The DIS architecture is designed exploiting the Grid Monitoring Architecture (GMA) approach proposed by GGF[12] and the Aggregator Framework software[13] framework distributed with the Java WS Core:

- GMA is an abstract design 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).
- The Aggregator Framework is a software framework on which it is possible to build grid services that *(i)* collect information from grid resources using pluggable aggregation sources, *(ii)* deliver collected information to pluggable sinks, *(iii)* and manage creation and destruction of individual aggregation registrations. It makes use of the following standards: WS-ResourceProperties (WSRF-RP), WS-ResourceLifetime (WSRF-RL), WS-ServiceGroup (WSRF-SG), WS-Topics, and WS-BaseNotification.

5.1.1.1 Service architecture

Six physical components implement the DILIGENT Information Service:

- **DIS-IP** (WSRFService) – The DIS-IP is a web service that is deployed on each DHN. It is responsible for gathering information about the resources stored on the node. Exploiting the Aggregator Framework capabilities, but hiding the complexity of the implemented specification, it provides support for the publication of information about the DHN, the resources, the services hosted on it, and any other entities considered as useful information. All the collected information is published to the appropriate DIS-IC instance as WS-Resource Properties. It is worth noting that each DIS-IP can also mirror each information by creating a copy of it on two or more DIS-ICs.
- **DIS-HLSClient** (Library) – The DIS-HLSClient is a library that is available on each DHN. It is used by the services hosted on the node to have access to the information

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

maintained by the DILIGENT information and monitoring service. This library enables locally hosted services to have access, as consumers, to the information while hiding details about information partitioning among instances. Actually, this library is designed to operate on locally available information that is built and maintained by the DIS-Cache service. The discovery of the information can be achieved through a proactive request, expressed by means of a query, or through a passive one, expressed by means of a subscribe request.

- **DIS-Cache** (WSRFService) – This service is in charge of building and maintaining a local image of the information that is globally available on the various DIS-IC instances. The information that is locally available represents the minimal subset of all the information that is needed to fulfill the information needs of the services hosted on the node. This service is also built by relying on the Aggregator Framework. Thus the DIS-Cache acts as an Aggregator Sink where the Aggregator Sources are the DIS-ICs needed to fulfill the node needs. It is up to this service to rearrange and reconfigure the framework by adding new Aggregator Sources in order to fulfill the dynamicity of the node information needs.
- **DIS-R-GMAClient** (WSRFService) – This service is in charge of harvesting resource information from the R-GMA Server it has been configured to interact with. This information is related to grid resources (CEs, SEs, I/O Servers, etc.) belonging to the grid infrastructure to be joined. Then, the gathered information is manipulated in order to make it compliant with the schema adopted in DILIGENT and then published via the DIS-IP.
- **DIS-IC** (WSRFService) – This service represents the main aggregator of the information produced by nodes locally. It collects information produced by single resources via the DIS-IP service and makes them available via the WSRF resource property mechanisms.

  In order to avoid problems related to centralized solutions, like single points of failure and scalability, it is designed to operate in a distributed and replicated way. In particular, each DIS-IC instance can be configured to collect information from an established set of information sources (Aggregator Sources in Aggregator Framework terminology). Moreover, as each DIS-IC publishes collected information as resource properties, it can also be considered as a kind of Aggregator Source and thus be used by another DIS-IC in a usual way. The DIS-IC also provides standard WSRF resource property query and subscription/notification interfaces to retrieve information. It is worth noting that the service is designed to forward queries and subscription requests to the appropriate DIS-IC service instance when it does not maintain the requested information.
- **DIS-Registry** (WSRFService) – This service provides registration and un-registration facilities for the DILIGENT resources profiles (i.e. the DILIGENT resource profile described in Section 3.1.2) as well as their storage and preservation.

By relying on top of the Aggregator Framework and the WS-ServiceGroup specification, all these components basically allow the DIS to manage only WS-ResourceProperties. When this is not possible the DIS provides a way to transform the information into WS-ResourceProperties. For example, the DIS-R-GMAClient has been designed to harvest information and republish its content as a WS-ResourceProperties.

5.1.1.2 Interactions

DIS-HLSClient

- The DIS-HSLClient is the access point to allinformation handled by the DILIGENT information and monitoring service. It serves all the components that need to retrieve such information for their purposes.

DIS-IP

- The DIS-IP is responsible for collecting status information from resource information providers located in the same DHN. Because of its being the only publisher available in the infrastructure, it is in charge of providing the collected information to the DIS-IC that thus represents a second level of aggregation. For this reason and in accordance to the Aggregator Framework, it plays both the role of an Aggregator Source and the role of an Aggregator Sink. It is an aggregator sink with respect to the resources hosted on the node, while it is an Aggregator Source with respect to the DIS-IC.

DIS-Registry

- The information provided at registration time of any DILIGENT Resource compliant with the DILIGENT Resource model presented in Section 3.1.1 needs to be transformed in a more suitable format manageable by the DILIGENT Information Service. To achieve this goal, the Registration/Unregistration Services interact with the DIS-Registry each time a new resource is registered passing the "profile" information of the resource. In this way the new available resources are discoverable by all the other components that access to the DILIGENT Information Service via the DIS-HLSClient.

DIS-R-GMAClient

- When collecting information about gLite resources, the DIS-R-GMAClient needs to interact with the CredentialMapper to obtain valid gLite credentials to have access to the RGMA Server service of the underlying Grid infrastructure
- Since an accessible gLite resource is also a DILIGENT Resource, the DIS-R-GMAClient interacts with the Registration/Unregistration Services to register them into the DILIGENT Information Service and into the VOs of the DILIGENT infrastructure.

5.1.1.3 Deployment scenario(s)

Figure 16 depicts a possible deployment scenario of the components forming the DIS. This scenario is based on the distribution of the information among the various DIS-ICs based on the concepts of DL and VO. In particular, in this scenario we decided to have three levels of aggregation, namely, per node, per DL, per VO. Per node aggregation is performed via the DIS-IPs services instantiated on each DHN. Per DL aggregation is performed by instantiating a DIS-IC service for each DL. This service is in charge of aggregating data coming from the DHN hosting the services that form the DL. The third level of aggregation, per VO aggregation, is performed by instantiating a DIS-IC service instance configured to collect data coming from DIS-IC associated with the DLs generated by the VO.

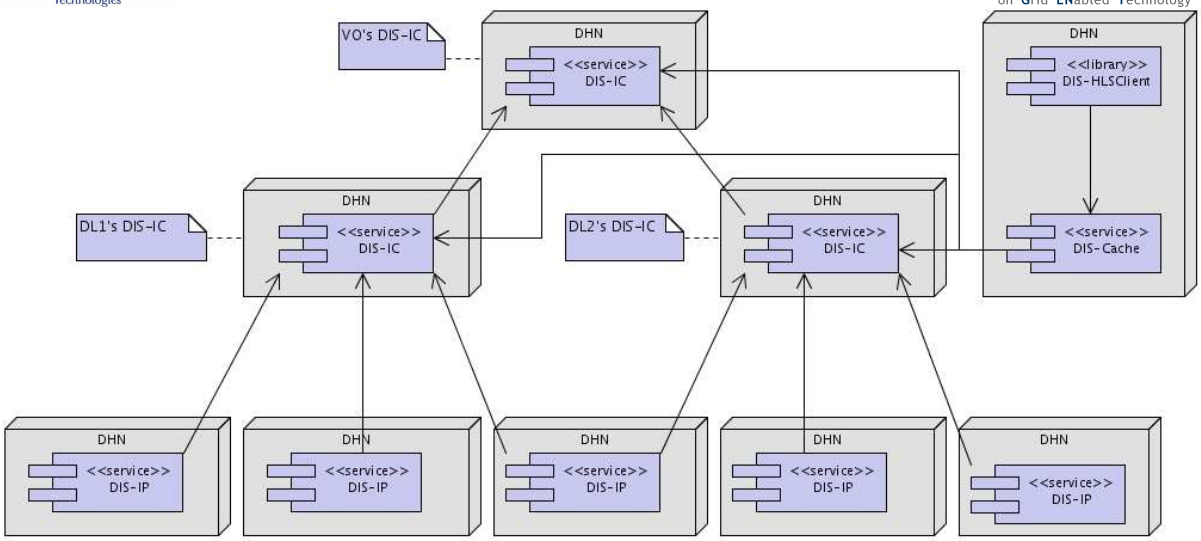


Figure 16. DIS – deployment scenario

5.1.2 Broker & Matchmaker Service

The Broker & Matchmaker Service (BMM) is in charge of supporting the Keeper service in deploying a new Digital Library on a set of DHNs. In particular, once the Keeper has identified the set of packages needed to build a new DL (more generally, any time it needs to deploy a new package, see Section 5.1.3) and their requirements and relationships, the BMM's task is to identify a set DHNs to be used as target hosts for the deployment.

5.1.2.1 Service architecture

From an architectural point of view, two components have been identified:

- **Matchmaker** (WSRFService) – This is a VO-specific component. To improve availability, multiple instances of this service can be deployed on different DHNs. The number of instances is determined at VO creation time by the Keeper according to performance and reliability needs. Each instance updates its own copy of the DHNs environment status using the DIS notification mechanism, thus ensuring the consistency of the knowledge used to drive the decisions. Failure feedbacks, conversely, are circulated using replicas within the federation, adopting a push and pull model.
- **BMM-API** (Library) – It provides local access to BMM functionalities, avoiding the need to select and contact a Matchmaker replica. This library is typically deployed together with the DL Management Service (see Section 5.1.3).

5.1.2.2 Interactions

Matchmaker

To update its own copy of the DHNs environment status, the Matchmaker subscribes itself via the DIS-HLSClient to receive notifications from the DIS about any change in the status of the DHNs forming the VO. For this purpose, it also collects feedback from the DL Management after the deployments.

5.1.2.3 Deployment scenario(s)

There is at least one Matchmaker service per each VO that maintains (in collaboration with the DILIGENT Information Service) an up-to-date picture of the DHNs status forming the VO. Replicas of it within the VO are allowed. Each replica has its own knowledge of the system derived from the information disseminated by the DIS. However, a periodic

synchronization among the replicas is also provided to exploit the feedback information they collect from the deployment activities.

5.1.3 Keeper Service

The Keeper is primarily responsible for the creation of a new digital library by instantiating its resources and by authorizing its users. It is also in charge of the deployment of all the packages needed for the management of a new VO, and of the registration of new DILIGENT resources. Finally, it must guarantee the overall set of the VDL functionalities at any time by dynamically relocating resources and periodically checking their status.

The DHN and software package are the two keys to achieve the dynamic deployment and, therefore, the dynamic creation of DLs. Thus, they also are the building blocks around which the Keeper is designed. In fact, in order to manage them, we need: *(1)* a location where to store the packages, *(2)* a local installer on each DHN, and *(3)* a manager that remotely coordinates the DL or the VO package deployments. Following this needs, the Keeper service has been decomposed in components whose behaviour is described in the next subsection.

5.1.3.1 Service architecture

The Keeper has been decomposed in the following four components:

- **Package Repository** (WSRFService) – It is a WSRF service that handles all the installable software packages. The Package Repository is the place where packages are stored. The uploading phase is managed by the RegistrationUI portlet and the RegistrationService service of the DVOS (see Section 5.1.4.5). These components upload a new package into the Package Repository that validates the package (e.g., as for its conformity to the Package Model) and returns a positive/negative response. Once a package is stored, the service must serve the DL Management and Matchmaker components in order to retrieve information about a package (mainly to check dependencies) and the Hosting Node Manager in order to move the requested packages on a DHN before their deployment. To deliver this latter functionality, the Package Repository relies on the GridFTP data transfer protocol.
- **DL Management** (WSRFService) – It is the WSRF service in charge of coordinating the deployment process of packages within a DL/VO and to respond to events that occur during the DL/VO life-cycle. In particular, after a new DL is designed, it has to:
    - manage the DL definition criteria passed by the VDL Generator service
    - manage services dependencies, requirements, and QoS parameters
    - interact with the Matchmaker to identify the appropriate DHNs where to deploy the identified services
    - interact with the Hosting Node Manager of the DHNs selected by the Matchmaker to start the deployment process
    - report feedback to the Matchmaker about the deployment activities

  During the DL lifetime, it has to:
    - coordinate and disseminate the operational context that transforms this set of distributed DILIGENT resources into a single geographically distributed application. In the DILIGENT terminology this context is named *DL Map*; it specifies the DL resources locations and their configuration; it is built exploiting the resource profiles maintained by the DIS-Registry. Any other dynamic information about a resource (e.g. its status) is maintained and disseminated by the DILIGENT Information Service.

- monitor a DL by:
  i. creating new Running Instances in order to maintain the required level of QoS
  ii. creating new Running Instances after a fault
  iii. responding to new DILIGENT resource registrations in the DIS

The DL Management is also involved in the process of the creation of a new VO performed by the DVOS. When a new VO is created:

- the AuthorizationService service notifies the DL Management
- the DL Management identifies which mandatory services have to be deployed for the management of the new VO (AuthorizationService, DL Management, Matchmaker and any other VO-mandatory labeled package)
- and, finally, the DL Management deploys these packages in collaboration with the Matchmaker and the Hosting Node Manager in the usual way

It is worth noting that using the same basic set of functionality the DL Management manages different aggregations of resources. Depending on the type of aggregation, named "scope" in this contest, a DL Management has a different behavior. There are two possible scopes: the *VO scope* and the *DL scope.* In the first case the goal is to manage the packages within VOs as well as the DL Management instances of the DLs belonging to the VOs. In the second case the goal is to manage packages for a specific DL. As a consequence there must be at least one DL Management for each DL and for each VO.

- **Hosting Node Manager** (WSRFService) – The Hosting Node Manager (HNM) is a WSRF service that must reside on each networked node that aims at hosting DILIGENT services, i.e. DHN. It is in charge to deploy new packages on the local node following the instructions of the DL Management. Moreover, when activated for the first time on a new node that joins the DILIGENT VO, it identifies the DHN-mandatory packages and deploys them on the node. The HNM interacts directly with the Java WS Core software and related tools and modifies the package files to reflect the configuration requested by the DL Management. The node management performed by the HNM involves the following tasks:
  - deploy and un-deploy new packages on the node
  - configure, activate, and deactivate packages on the node
  - maintain and expose the node configuration to the deployed services
  - exchange data with the DL Management component
  - exchange data with the deployed services
  - push DHN and deployed service configurations to the local DIS-IP
  - push DHN status information to the local DIS-IP
- **K-UI** (Portlet) – It is a portlet that provides a graphical user interface accessible through the DL portal. It permits the execution of administrative tasks and guarantees the access to a report of the DL status.

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

5.1.3.2 Interactions

DL Management

- The DL Management manages the deployment process of the software packages on the DHNs. It interacts with the BMMClient to obtain a list of suitable DHNs where it

deploys the requested packages. After the deployment operations, it reports feedback to the Status&FeedbackCollector that takes care of this information for future requests.

- The DL Management interacts with the AuthorizationService to verify the necessary authorization rights for the deployment actvivities. It is also notified by the AuthorizationService about the new VOs available to start the deployment of the mandatory packages.
- The DL Management interacts with the Registration/Unregistration and the DIS-Registry services to register the deployed services as new DILIGENT Resources. This operation is performed using the automatic registration/unregistration modes provided by these two services.
- Finally, the DL Management interacts with the DIS-HLSCLient to retrieve information about the locations and status of the other DILIGENT services and resources.

Hosting Node Manager

- The Hosting Node Manager publishes the status of the DHN it manages in the DILIGENT Information Service by interacting with the DIS-IP. The status includes data collected from the local running instances; these data are not related to any particular WS-Resource created by a service
- The Hosting Node Manager also publishes the DHN profile and the profile of each activated package via the DIS-IP. It also supports the updating of the component profile that can be requested by the correspondingly running instance.

5.1.3.3 Deployment scenario(s)

Figure 17 illustrates a typical deployment scenario of the Keeper components within a DL. Here, a DL Management interacts with a Package Repository and several Hosting Node Managers to deploy the software packages needed to set up the DL.Oncethe packages are installed, on each DHN the HNMs interact with the local deployed services to gather information to publish in the DIS.

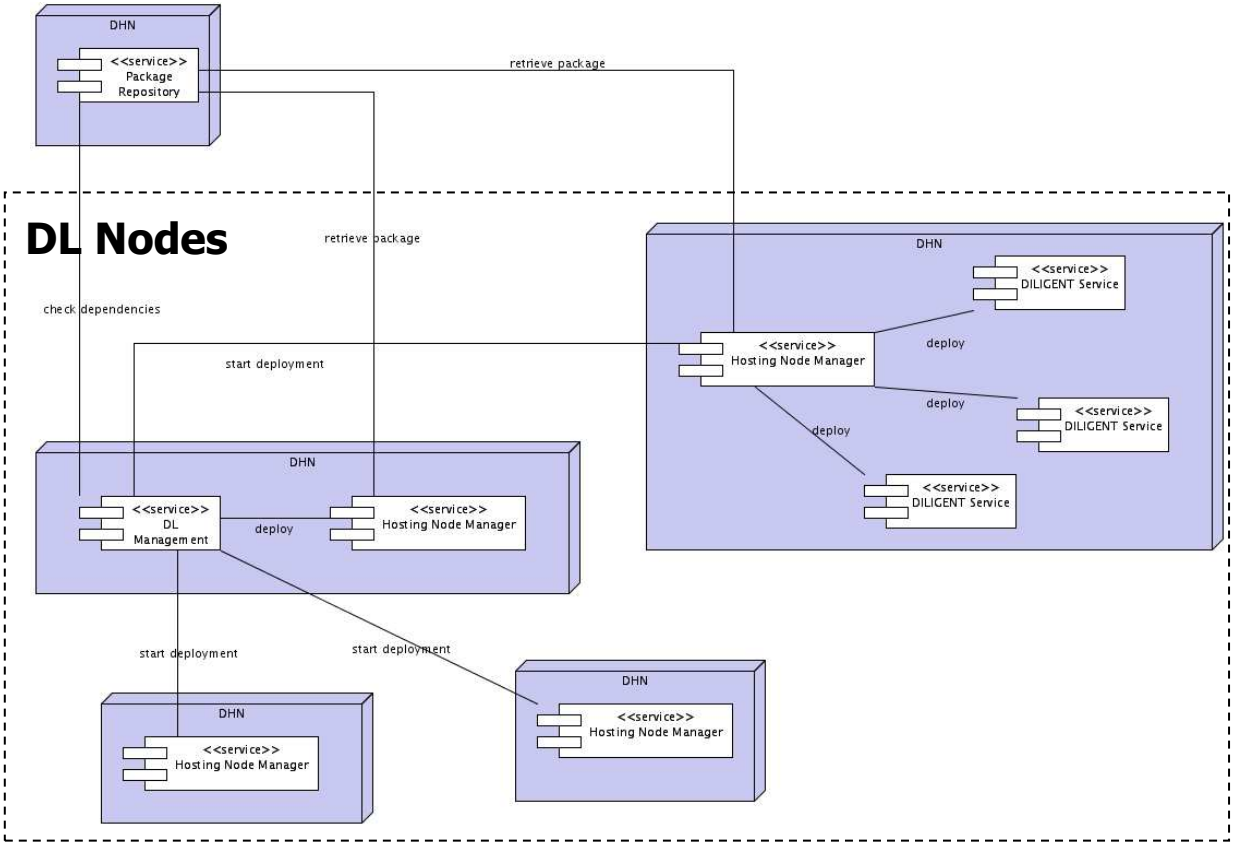


Figure 17. Keeper – deployment scenario

5.1.4 Dynamic VO Support Services

5.1.4.1 Authentication Support

Authentication support deals with user identity and credentials management issues. The DILIGENT Authentication model is based on the Grid Security Infrastructure (GSI). This implies the adoption of a Public Key Infrastructure (PKI) and the usage of X.509 End Entity Certificates (EEC) [28] released by a Certification Authority (CA). GSI also supports delegation and single sign on in a distributed environment exploiting X.509 Proxy Certificates (PC).

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

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

The static assignment of EECs to Service Containers implies that a copy of credentials used by these entities must be available on the machine hosting the Container.

The dynamic assignment is available for dynamically deployed services. A new X.509 certificate is generated at the end of the deployment process and provided to the service during its initialization. This certificate can then be used by the service to authenticate itself to the other entities.

The main functionalities offered by this area are:

- *End Entity Certificate Issue*: issue of a new EEC and private key representing a DILIGENT identity. A trusted CA provides this functionality. This is the first step of the 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 by a machine where the EEC and the original private key of the user are available.
- *Credential Retrieval*: functionality used by DILIGENT Services (particularly the DILIGENT portal) in order to retrieve a short-term copy of the DILIGENT User credentials. 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 the main issue in DILIGENT authentication; the need of this functionality depends on Certification Authorities acknowledged by DILIGENT platform. If these Certification Authorities are also acknowledged by external[14] gLite infrastructures accessible by the DILIGENT platform, then credential mapping can be avoided. Otherwise, the mapping is always required in order to access those external gLite infrastructures. In this case, the entire DILIGENT VO could share the same gLite account in order to access an existing gLite infrastructure. Moreover, the mapping does not have to be static and can be modified through new agreements between DILIGENT communities and gLite resources owners.

5.1.4.1.1 Service Architecture

The Authentication Support package is decomposed in the following components:

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

5.1.4.1.2 Interactions

CredentialMapper

It is built on top of the VOMS Server of the underlying grid infrastructure in order to exchange credentials that allow DILIGENT Running Instances to access to the gLite services.

[14] The "external" attribute is used to identify gLite infrastructures authorizing VOs different from the DILIGENT one (e.g.: EGEE Pre-production service).

AuthenticationAPI

The AuthenticationAPI is the main entry point all the authentication functionality. It used by other components when they need to verify the identities and credentials of their callers.

5.1.4.1.3 Deployment scenario(s)

The CredentialMapperUI can be deployed in one or more DILIGENT portals. The AuthenticationAPI library must be deployed in all hosting nodes. The CredentialMapper is a centralized service but can be replicated in more then one hosting node to improve the reliability of its functionality.

5.1.4.2 Authorization Management

Multiple levels of authorization can be defined for DILIGENT Resources. This package provides support for VO-level Authorization. It originates from the VO model as defined in Section 3.2. Service designers can also define and use other finer-grained levels of authorization.

The DILIGENT authorization model relies on the underlying Java WS Core authorization framework. It enables DILIGENT Services to separate security issues from service specific behaviour. This is achieved using a chain of authorization handlers managed by the Service Container. These handlers are used during evaluation of incoming requests in order to permit or deny access to a service. The authorization chain can be configured through an XML file named "Security Descriptor"[15]. Each DILIGENT Service is in charge to implement these authorization handlers in order to enforce VO-level and resource-specific authorization policies.

5.1.4.2.1 Package Architecture

Each component is briefly described afterward.

- **AuthorizationUserInterface** (Portlet) – In order to access and manage a VO AuthorizationService a corresponding AuthorizationUserInterface portlet must be available in a Web portal. In particular:
    - the DL Web Portal hosts this portlet for the VO modelling the DL
    - for other VOs (e.g.: ARTE VO, ImpECt VO), the DILIGENT Web Portal hosts the corresponding portlets.

  The AuthorizationUserInterface uses functionalities provided by the Authorization API library.
- **AuthorizationAPI** (Library) – this library hides to clients (e.g.: DILIGENT Resources, AuthorizationUserInterfaces or authorization handlers) the internal structure of the Authorization Management package. It allows clients to easily contact AuthorizationServices. This library is deployed on each DHN including portal nodes.
- **AuthorizationService** (WSRFService) – this is a WSRF Service representing a single VO. A new instance of this service is deployed when a new VO is created. These services are in charge to maintain the authorization status of the DILIGENT VO hierarchy. Its status can be inspected and modified through functionalities of the AuthorizationAPI library.

5.1.4.2.2 Interactions

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

AuthorizationService

Each AuthorizationService represents a single VO in the DILIGENT VO hierarchy and, therefore, it supplies a "VO-local" view of this information to the components that ask for their own authorization rights. Information returned by an AuthorizationService is related to the position of that VO in the hierarchy. Within the Collective Layer, this includes interactions with the DL Management, the DIS-HLS and the NotificationDispatcher via the AuthorizationAPI.

AuthorizationAPI

This library is used by any DILIGENT service that needs to verify the authorization policies.

5.1.4.2.3 Deployment scenario(s)

The AuthorizationUserInterface can be deployed in one or more DILIGENT portals. The AuthorizationAPI library must be deployed in all hosting nodes. The AuthorizationService is VO specific and must be present for each VO and for each DL.

5.1.4.3 Notification Support

The aim of the Notification Support is not to provide a general mechanism to manage events occurring in the system. Such a general mechanism is already provided by the WS Notification [29] implementation of the Java WS Core. Rather, the twofold purpose in designing the DILIGENT Notification Support is to allow users (or roles) to be informed about relevant[16] changes in the system and suggest to these users options available as a consequence of these changes.

5.1.4.3.1 Package Architecture

Notification Support relies on three distinct components that can be deployed independently on different DHNs:

- **NotificationUI** (Portlet) – This enables DILIGENT users to change their own delivery model (push/pull mode, email delivery, frequency of delivery, etc.)
- **NotificationService** (WSRFService) – This WSRF service is in charge to provide functionalities 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).
- **NotificationAPI** (Library) – This library provides a uniform access to the Notification Service functionalities.

5.1.4.3.2 Interactions

NotificationUI

This portlet interacts with the UserGroupAPI (see Section 5.1.4.4) in order to store the dispatching model in the profile of the user.

NotificationService

When a notification is not referred to a single user, the NotificationService invokes the AuthorizationAPI library to query group members and users with a given role in order to know which users have to be notified.

5.1.4.3.3 Deployment scenario(s)

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

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 provides them with an endpoint reference to the NotificationUI for retrieval and configuration. The NotificationUI can be deployed in one or more DILIGENT portals. The NotificationAPI library must be deployed in all hosting nodes.

5.1.4.4 User and Group Management

According to the D1.1.1 "Test-bed Functional Specification" **Erreur ! Source du renvoi introuvable.**, the management of information about users and group of users is mainly divided into four functional parts: *users management*, *groups management*, *search*, and *invitation.*

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

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

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

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

The *Invitation* part deals with invitee users and groups to join a DL, to get access to a Collection, etc. The mechanism is similar to the one adopted in the invitation of a user to a group.

5.1.4.4.1 Package Architecture

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

- **InvitationService** (WSRFService) – It is a WSRF service that can serve multiple VOs. Multiple instances of this service can be deployed in order to increase availability.
- **ProfileRepository** (WSRFService) – It is 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** (Portlet) – It is a portlet component and is hosted by the DILIGENT Portal Engine.
- **UserGroupAPI** (Library) - This library provides a uniform access to the User and Group Management functionalities.

5.1.4.4.2 Interactions

Profile Repository

The ProfileRepository interacts with the AuthorizationManagement 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.) by means of the AuthorizationAPI.

5.1.4.4.3 Deployment scenario(s)

The UG-GUI can be deployed in one or more DILIGENT portals. The UserGroupAPI library must be deployed in all DHNs. The InvitationService can be deployed in one or more hosting nodes. Finally, the Profile Repository is VO specific and must be replicated for each VO and for each DL.

5.1.4.5 Resource Registration Support

Resource Registration Support provides mechanisms enabling resource owners to share their resources with the DILIGENT community. The DILIGENT infrastructure imposes several constraints on resource usage, where the most important is the conformity to the DILIGENT Resource Model presented in Section 3.1.1 and the fulfillment of some sharing policies. Both registration and the definition of such sharing policies are managed through the RegistrationUI by *Resource Managers*. Examples of resources that can be registered are DHNs and software packages.

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

- a *profile* – an XML description of the resource conforming to the DILIGENT Resource Model (see Section 3.1.2)
- a set of *sharing rules* – rules that define the way a VO or a user is entitled to use a resource.

5.1.4.5.1 Package Architecture

The Resource Management components are:

- **RegistrationUI** (Portlet) – It is a portlet that allows Resource Managers to manually register (and unregister) resources by interacting with the RegistrationService and UnregistrationService.
- **SharingRulesUI** (Portlet) – It is a portlet that provides Resource Managers with functionalities to request inclusion of resources into VOs and to modify VO Sharing Rules.
- **RegistrationService** and **UnregistrationService** (WSRFService) – They are WSRF services that provide registration and unregistration functionalities.

5.1.4.5.2 Interactions

Registration/Unregistration Services

They provide a way through which the DILIGENT Resources can be registered/unregistered in the DILIGENT infrastructure. This includes interactions with:

- the AuthorizationAPI, for the arrangements of the permissions on the resources
- the DIS-Registry, for the registration of the resource profiles in the DILIGENT Information Service
- the Package Repository to store software packages (in the case of registration/unregistration of this kind of resource)

5.1.4.5.3 Deployment scenario(s)

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

5.1.5 VDL Generator Service

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

- supports users/communities in specifying the criteria that characterize the new DL, e.g. the information space, the required functionalities, the characteristics of these functionalities;
- selects the appropriate pool of services and information sources required to implement a DL that satisfies the specified criteria;
- notifies to the DL Management service (see Section 5.1.3) the identified services.

This service is also characterized by a high interaction with a user via a graphical user interface. When specifying the architecture of such kind of interactive service, the challenge is to keep the functional core independent from the user interface. In fact, while the core is based on the functional requirements and usually remains stable, the user interface is often subject to changes and adaptation. This base design choice has consequences on the entire design of the VDL Generator Service.

5.1.5.1 Service architecture

The architecture of the VDL Generator Service is based on a three-layered configuration:

- ➔ the *front-end* is mainly formed by the components that provide the presentation layer of the service, i.e. the VDLGeneratorView and the VDLGeneratorController. These components implement the logical classes having the same name;
- ➔ the *middle-tier* is composed by the logic of the VDL Generator Service, i.e. the VDLGeneratorModel, the SystemKB and the VDLReasoner. For performance reasons the latter two components are deployed on the same node where the VDLGeneratorModel service is deployed;
- ➔ the *back-end* is the storage component in charge to maintain the VDL Definitions.

Therefore, the VDL Generator service is composed by:

- **VDLGeneratorModel** (WSRFService) – The VDLGeneratorModel service represents the access point to the VDLGeneratorLogic functionalities. As a consequence, it is the service in charge to offer the core functionalities of the VDL Generator Service. It is implemented as a WSRF service and we assume that at least one instance of this service is always available within the DILIGENT infrastructure.
- **VDLGeneratorController** (Library) – The VDLGeneratorController is in charge of handling the inputs provided by the DL Designer via the VDLGeneratorView portlet. This component has been designed as a library and it has to be deployed on the same DHN where the VDLGeneratorView is hosted by the DILIGENT Portal.
- **VDLGeneratorView** (Portlet) – The VDLGeneratorView is designed as a portlet that allows the DL Designer to create new VDL Definitions.
- **VDLDefinitionRepository** (WSRFService) – The VDLDefinitionRepository is a storage manager for the VDL Definitions defined by DL Designers.

5.1.5.2 Interactions

VDLGeneratorModel

- To compile the list of available users and DILIGENT resources (allowing the DL Designer to define the characteristics of a new VDL), the VDLGeneratorModel interacts with the DIS-HSLClient (5.1.1), the SearchService (5.3.1) and the UserAngGroupClient (5.1.4.4)
- Once the DL Designer is ready, the service interacts with the DL Management service (5.1.3) in order to pass the definition criteria of the new VDL
- To send invitation to users to join the new VDL, the service interacts with the NotificationClient (5.1.4.3).

5.1.5.3 Deployment scenario(s)

The deployment scenario of the described components is quite simple because the VDL Generator can be considered as a centralized component with respect to the entire infrastructure. It is not expected to have a number of replicas of the service for performance reasons since there are no strong requirements in terms of response time.

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

5.2 Content & Metadata management services

The *Content & Metadata management* services provide access and storage of information objects, management of related metadata, encryption and decryption of information objects for security reasons, and information object annotations management.

As a consequence, the services constituting this area can be grouped according to these four functional areas:

- *Content Management* offers functionality for accessing and storing information objects by relying on (i) local storage capacity, (ii) storage capabilities offered by a gLite based grid infrastructure, and (iii) addressing objects stored by third party information sources (via appropriate wrappers). In the case of storage, data are distributed and replicated among the allocated storage nodes to increase the reliability and the performance of the system. The distribution schema and the real storage locations are hidden for the users.
- *Metadata Management* offers functionality for associating to and managing metadata about information objects. Besides offering basic functions like insertion, deletion, and update, this service takes care of managing mapping between different schemes of metadata and is in charge of importing the related metadata when an external information source is imported into DILIGENT.
- *Content Security* offers functionality for supporting the encryption/decryption of the data as well as functionality to detect changes for most object types by watermarking the content. In fact, even if the content objects are distributed over the nodes of trusted organizations, users must be certain that unauthorized actors will not change their content. In the extreme case, this means that nobody outside the trusted boundary of the digital library or virtual organization is able to even read the content.
- *Annotation Management*, a service in charge to offer interactive functionality for the management of manually authored, subjective, and context-dependent metadata items about information objects.

From an architectural point of view these groups of services are organized into a layered architecture (Figure 18) constituted by three layers: the Base Layer, the Storage management Layer, and the Application Layer.

The **Base Layer** is an abstraction from basic Grid storage facilities, as found in existing Grid infrastructures. Concretely, the Base Layer provides an installation of *gLite I/O Server* and the *FiReMan repository*.

The **Application Layer** is the topmost one and contains the services that the other DILIGENT services can rely on, i.e. the *Content Management* and *Metadata Management* services that expose interfaces for (*i*) document storage, (*ii*) notification management, (*iii*) collection management, (*iv*) metadata associations, and (*v*) information sources import.

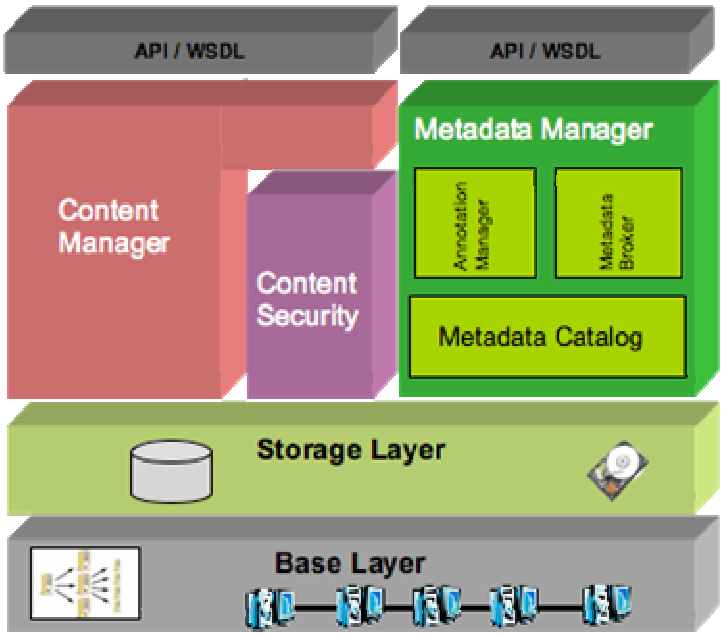


Figure 18. Content & Metadata Management: Layered Architecture

As depicted, the **Storage Layer** is wrapped around the Base Layer (that includes also the gLite I/O capabilities) and takes responsibility to (1) introduce information objects as an abstraction for manifold content, (2) associate storage properties to information objects, (3) introduce generalized relationships between documents, (4) preserve bindings to file-based documents, (5) a basic service interface for querying and updating the content, its associated storage properties and relationship information, (6) maintain consistency of object relationships through appropriate database constraints (foreign keys, no cascading deletes), (7) provide transparent data replication with query routing to the appropriate node, and (8) update propagation and guaranteed freshness in querying the data. Notice, that the Storage Management Layer is not aware of any semantics that come with information objects. That is, metadata bindings (a special form of an object relationship) are made persistent in the Storage Management Layer.

Storage Management Layer builds on top of (1) the Base Layer and (2) a number of relational database system instances that are distributed on storage nodes. In detail, every storage node will host (at least) one database instance with the schema as depicted in Figure 19. Figure 19 is an ER schema, which represents the information object type as introduced earlier:

- Each information object may comprise a list of storage properties represented as simple key-type-value associations. Those storage properties are atomic whereas complex metadata (like indexes, multimedia features) may also be represented as separate information object that is associated to the object. Plain tags should be flavored, whenever querying is a requirement or attribute-level access to certain metadata fields shall be provided within the storage manager.
- An information object may be an abstraction from a file-based document (which is stored in gLite I/O components), a BLOB-field representation of content, a simple document with no associated content (besides storage properties and/or object references), and a complex object that references to other objects. Notice, that complex documents are permitted to have content (file, BLOB) associated to it. The storage manager does not maintain consistency of the document content and explicit references to other objects. For instance, an HTML document that includes a number of images may be modeled as a complex object that provides references to information objects (containing the images). Alternatively, any XML (also HTML)

document can be made persistent within storage management without having the complete XML file associated to the respective information object as binary content. To do so, one can exploit the relationship attributes on role (of the relationship), position (of the contained object within the containing object), and cardinality. The advantage of having no duplicate association information (implicit to Storage Management in the XML file and explicit as persistent object relationship) comes at the price of providing parsers for XML documents (and possibly also other container formats) that break up and XML structures into their parts.

- An object reference "links" two information objects. Each object may (1) reference many other objects and (2) may be referenced by many objects (m-n relationship). To instantiate on this generic object reference, role names, reference types, a position tag (to reconstruct a view of the referring object by composing it out of its referenced objects), and a delete propagation rule (consulted upon removal of the referring object to determine whether to automatically delete the referred object) attribute the link. For instance, a reference type may be "indexes" with a role name that gives additional information, like "full-text index". In addition, a positioning attribute helps in representing an object that references to other objects, like an aggregate made up of components that have to be fitted together in a certain order. Propagate role provides removal constraints to the storage manager (i.e. the Storage Management service). In detail, different reference types may impose different rules when deleting a referenced object. For instance, component objects (appearing as collection members) will not be deleted if the collection is removed. In contrast, an associated index will be removed.

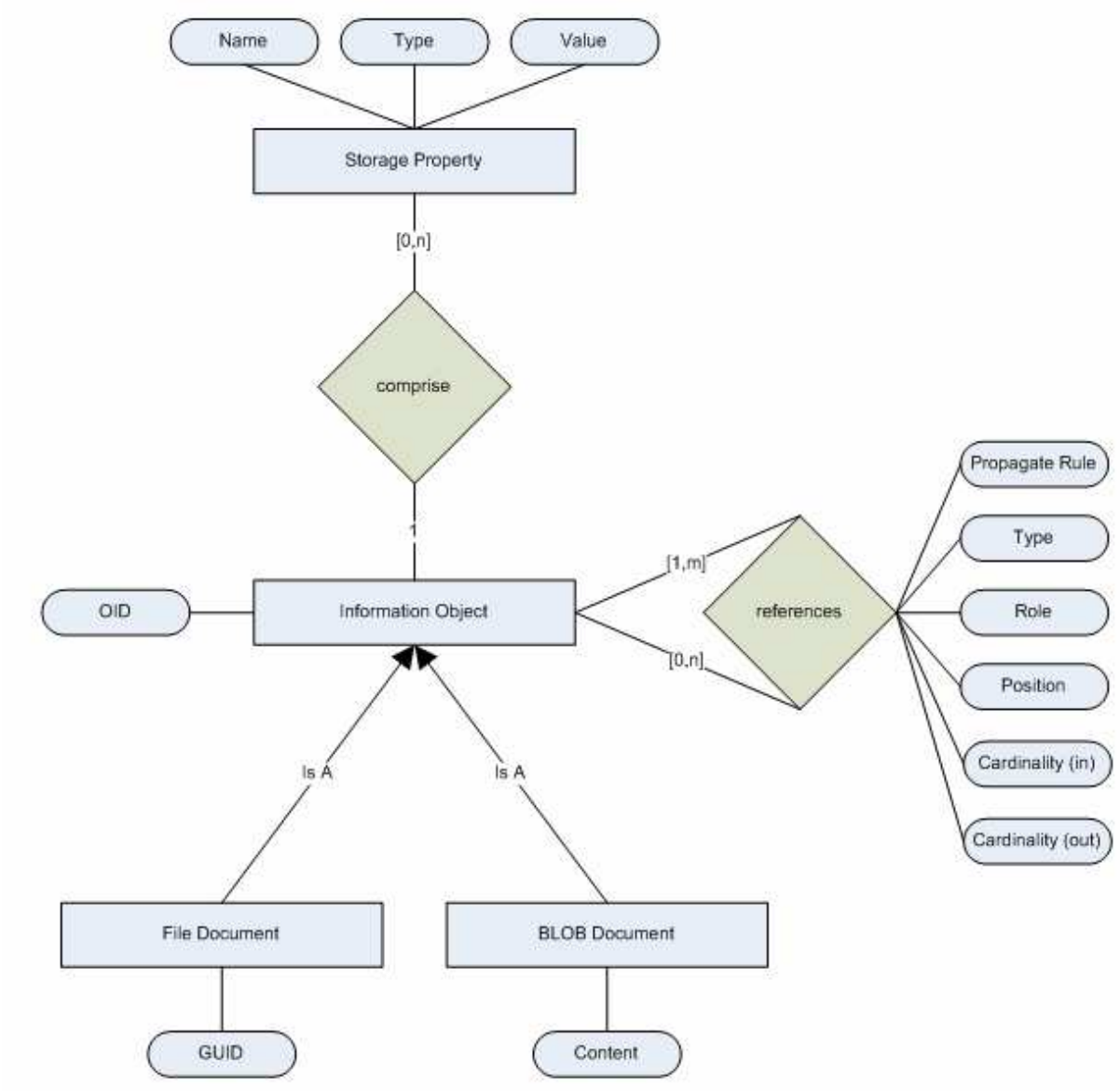


Figure 19: ER Schema of Proposed Object Model

Storage Management maintains so-called "storage properties" which is a form of plain storage-related attributes Its purpose is to keep track of additional information about an information object which is used for the internal use of storage management, like optimizing accesses etc. For the time being, we have identified the following storage properties:

- **Owner** identifies a DILIGENT user who owns that information object. Typically, the user who has created an information object becomes the owner.
- **Permission** is plain access/update/removal directive for non-owning users. It states whether other users may access (read), update or remove this information object. It has been introduced to support a plain security mechanism.
- **Type** identifies the kind of information object, like document, collection, aggregate, metadata, external document, service etc. It is important in replication and various Content Management functionalities.
- **Availability** is a constraint that guidelines the replication strategy of Storage Management. It tags the priority of a document, which will be consulted to decide how often an information object must be replicated. In other words, it is a qualitative information on the "importance" of a piece of content.

- **Name** is a plain string that can be attached to an information object for ease of identification. It may be employed to additionally equip information objects with non-unique, yet human-interpretable names.
- **URL** is an access pattern for external documents that physically reside in archives and are only reflected by a placeholder information object in storage management. It is essential for archive import, in particular, if those archives host content which is not imported into DILIGENT storage but resides in the archive.
- **Service** is the name of a "callback" service that will be invoked (if registered) upon changes in an information object (notification service). In other words, the notification service relies on this information to pass the information upon a changed object to the interested party.
- **Membership Predicate** is a Boolean expression that is used to initially (permanently) populate a materialized (virtual) collection, which is a special kind of aggregate object. It is, thus, essential for collection support.

Figure 20 depicts the overall and integrated picture of the Content & Metadata Management components in which such services have been designed and their relationships. The next sections present each component, its interactions and some possible deployment scenarios.

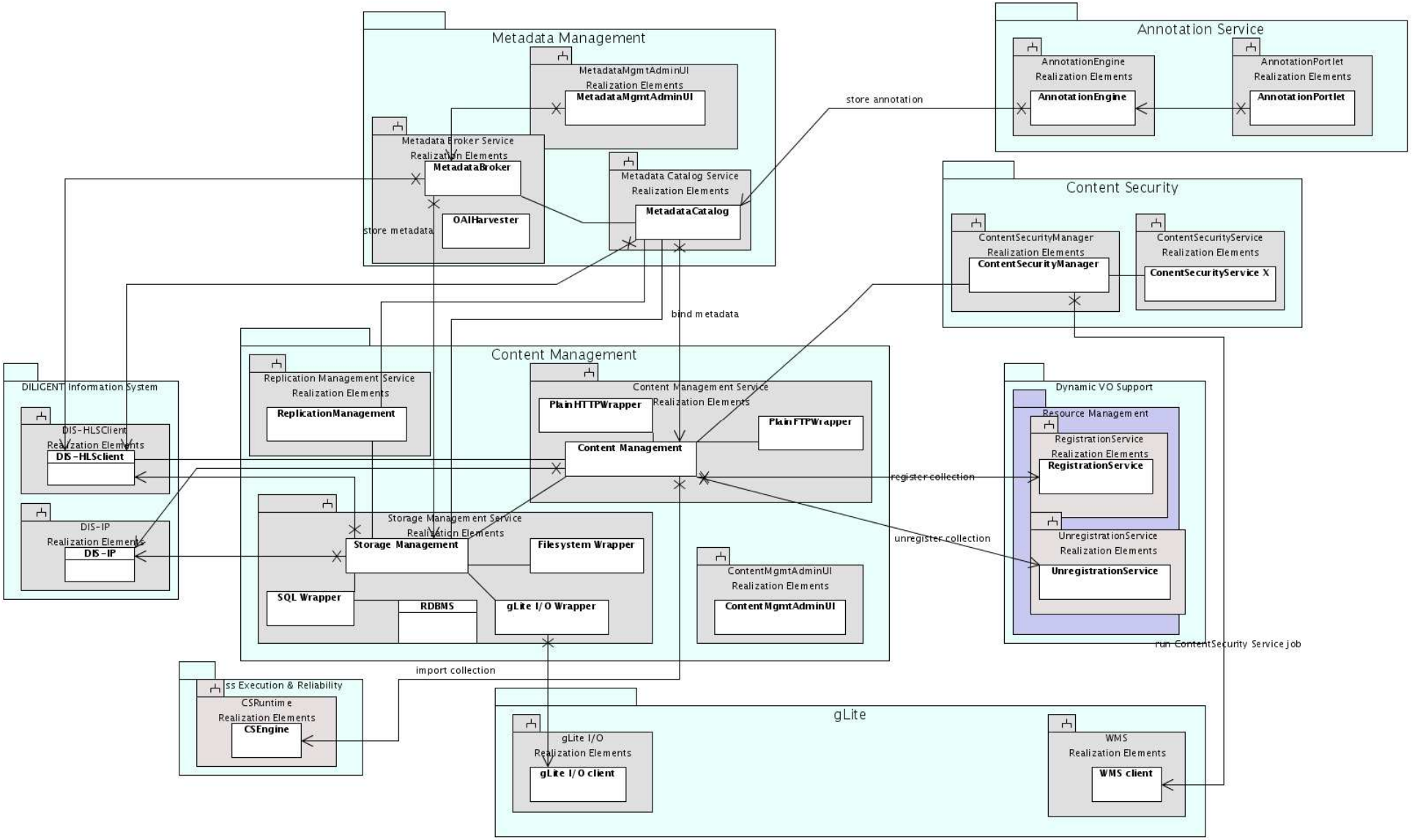


Figure 20. Content & Metadata Management services – architectural view

5.2.1 Content Management Service

The Content Management service provides operations for persistently storing, physically structuring, and efficiently fetching any kind of content on the Grid. Its requirements go clearly beyond file-system-like functionalities as natively provided by gLite. In particular, certain "flavors" of content must be distinguished, content must be interrelated in multiple ways, and content must be attributed by arbitrary properties. In addition, content may reside in external data sources that must be coupled to DILIGENT. And, finally, Content Management must provide appropriate means of observing content for changes and propagating those events to interested parties. It addresses services from the Content Management (Service) Layer, the Storage Layer and the Base Layer. In detail, we discuss the Content Management service, the Storage Management service, and the Replication Service, as part of the Base Layer (Figure 18).

5.2.1.1 Service Architecture

The design solution of the Content Management Service is composed by the following components:

- **Content Management Service** (WSRFService) – This WSRF service provides the high-level operations that are mapped into generic Storage Management operations. These operations are: (*i*) basic documents storage, access and update, (*ii*) collection management, (*iii*) fundamental metadata association, (*iv*) import of external information sources and archives, and (*v*) notification management. This service exposes each collection, document, and archive as a WS-Resource. The states of the object are an object's attributes and references and it will persist its state in the database. For example, states of a collection are its name, all its storage properties, and its references.
- **Storage Management Service** (WSRFService) – This service provides the basic operations atop our storage model for inserting, manipulating, fetching, and deleting information objects as being a generic abstraction of different concrete object types. That includes the assignment of storage properties and the setting up of inter-object relationships. It also provides operations for associating information objects with file-base documents stored in gLite storage elements. The Storage Management service exposes a web service interface for manipulating the underlying database. For moving files we use the GridFTP protocol. The Storage Management service relies on the DILIGENT Information service in order to discover the other instances of Storage Management services. In fact, it is worth noting that Storage Management Service instances do not act as independent entities but are interrelated by replication dependencies. That is, any update of an object must be propagated to all instances holding a replication of that object. Vice versa, one single instance may not suffice to answer a query which must be routed to another instance or be jointly processed by multiple instances.
- **Replication Management Service** (WSRFService) – An internal service which does not expose any interface to other services outside the Content Management. It is responsible for managing replications. It has been included to illustrate the basic principles of replication and guarantee freshness of querying data in DILIGENT.
- **ContentMgmtAdminUI** (Portlet) – Since Content Management is not supposed to provide a user interface for its major purpose (storage and retrieval of content), this portlet provides functionality for administrative tasks.

5.2.1.2 Interactions

Content Management Service

It interacts with:

- the Storage Management in order to perform the basic actions needed to provide the high level expected functionality. For instance, to return an Information Object it needs to access the object that is stored under the control of the Storage Management; to store an Information Object it needs to ask for the save functionality of the Storage Management, and so on.
- the external information sources via the wrapper components it is equipped with, namely the PlainHTTPWrapper and the PlainFTPWrapper.
- the DIS-HLSClient in order to discover the instances of the appropriate Storage Management Service.
- the DIS-IP in order to publish WS-ResourceProperties about *Collections*, *Archives*, *Observers*, and *Contents* (used for example by the Feature Extraction Service).
- the RegistrationResource and UnregistrationResource services of the Dynamic VO Support (Section 5.1.4.5) in order to manage the creation and deletion of a *Collection* as a DILIGENT resource;

Storage Management Service

It interacts with:

- the DIS-IP in order to publish on the DIS the WS-ResourceProperties of the information objects.
- the DIS-HLSClient in order to discover other instances of the Storage Management Service available in the infrastructure. In fact, the Storage Management Service is designed to operate in a replicate way, i.e. each instance communicates with other instances to implement replication strategies.
- the Replication Management Service in order to implement the replication strategy.
- the gLite I/O Client in order to have access to the storage resources of a gLite based infrastructure.

Replication Management Service

It interacts with other Replication Management Service instances in order to implement the replication strategy.

ContentMgmtAdminUI

It interacts with the Storage Management Service in order to perform administrative tasks, e.g. guiding the replication strategies, manually changing the replication performed.

5.2.2 Metadata Management Service

The overall purpose of the Metadata Management service (MMS) is to provide basic functionality for managing all aspects related to metadata: importing, transforming, storing, associating, deleting and updating metadata for information objects. It is also responsible for associating and maintaining the metadata information on the different kinds of DILIGENT Resources, like services and collections. The metadata managed by MMS are needed by many other services (Annotation, Content Description & Selection, Search, Index, etc).

To achieve the aforementioned functionality the Metadata Management treats the metadata objects as information objects; therefore they are handled by the Content Management Service. As a result of this design choice, the actual storing, maintaining and replication aspects of the Metadata Management Service are performed by relying on this service. Actually, the Metadata Management uses the functionality of the Storage Management

Service to make the right associations e.g. create the corresponding relationships between the different kinds of information objects, which represent thereby the information. Therefore the Metadata Management Service is tightly connected and related to the Content Management service, because it uses its operations to store and retrieve both the relationships and the metadata.

5.2.2.1 Service architecture

The Metadata Management Service is composed by the following three components:

- **Metadata Catalog Service** (WSRFService) – The Catalog WSRF service is responsible for managing the incoming requests for changing, accessing, manipulating metadata. It associates metadata with information objects, removes associations, returns metadata for a given object and updates associated metadata. From the WSRF point of view, this service can be seen as stateless, because it delegates the responsibility for the management of its states to other components. Data managed by MMS is stored in the persistence layer for data managed by the Storage Management service.
- **Metadata Broker Service** (WSRFService) – The Broker provides functionality for importing and transforming new metadata and thereby also managing the involved transformations. To be able to perform this task, the service offers management capabilities to store, associate, find and delete transformation rules, which can be applied to transform metadata from a source to a target schema. Additionally, transformation programs must be managed and maintained. They can be used to perform a series of transformation processes.
- **MetadataMgmtAdminUI** (Portlet) – Metadata Management is only working in the background and it is called by other services. However, we equipped the service with this portlet providing administration and configuration functionalities.

5.2.2.2 Interactions

Metadata Catalog Service

The Metadata Catalog Service interacts with:

- the DIS-HLSClient, to discover the instances of the Storage Management Service and of the Content Management Service it needs to interact with
- the Content Management Service, to manage associations between metadata and information objects
- the Storage Management Service, to physically store and retrieve metadata objects
- the IndexGenerator, to ask for generating new indexes or updating existing indexes

Metadata Broker Service

The Metadata Broker Service interacts with:

- the Metadata Catalog Service, in order to store the newly generated metadata and associate them with the appropriate information object
- the Storage Management Service, in order to physically store and retrieve the transformation rules

MetadataMgmtAdminUI

The MetadataMgmtAdmminUI interacts with:

- the Metadata Broker Service, in order to configure it (e.g. to update the transformation rules)
- the Metadata Catalog Service, in order to perform administrative tasks (e.g. tuning the interaction with the Storage Management Service instances)

5.2.3 Content Security service

Content Security Service is a part of the Content Management Service. It acts on the backend and it is not accessed by services outside the Content and Metadata Management area. The goal of this service is to secure the content stored by DILIGENT against unauthorized access via digital watermarking and partial encryption technologies. The former solution resides in enforcing copyright and integrity-checking of the media while the latter is a special type of encryption, i.e. a process altering the media quality without its full loss. Detailed information is out of the scope of this report, they can be found in [18].

5.2.3.1 Service Architecture

The complete Content Security service is composed by two components:

- **ContentSecurityManager** (WSRFService) – The Content Security service is organized in master-worker architecture and the ContentSecurityManager acts as the master. This service (i) analyzes the media content to be secured, (ii) splits it into expedient and independent chunks, (iii) distributes the chunks to a set of ContentSecurityServices performing the security tasks, and (iv) collects the single secured chunks to organize them into a whole bundle. The chunks are transferred to idle worker nodes of the system where the ContentSecurityService examines them.
- **ContentSecurityService** (GridJob) – The ContentSecurityService applies the security techniques to the chunks and sends them back to the ContentSecurityManager. It implements different algorithms implementing different mechanisms for diverse media. Then ContentSecurityService examines the media unit and then a selected algorithm is executed. After finishing the content security algorithm the media chunks are transferred back to the request ContentSecurityManager instance.

5.2.3.2 Interactions

ContentSecurityManager

It interacts with:

- the Content Management Service, in order to store the encrypted data
- the DIS-HLSClient, in order to discover which gLite WMS instance(s) it can access and use

ContentSecurityService

The ContentSecurityServices are executed as grid jobs on the gLite based infrastructure. Therefore the ContentSecurityServiceManager interacts with the WMS of the underlying gLite-based infrastructure discovered by interacting with the DIS-HLSClient.

5.2.4 Annotation Service

The DILIGENT Annotation Service (AS) offers interactive functionality for the management of manually authored, subjective, and context-dependent metadata items about DILIGENT information objects, primarily multimedia documents. In doing so, the AS mediates between DILIGENT applications and the Metadata Management Service (MMS). Actually, it builds over the functionality provided by the Metadata Catalog to provide information objects with annotations facilities. Details on the internal organization of this service and about its expandability are provided in [18].

5.2.4.1 Service architecture

The service is composed by two components:

- **AnnotationEngine** (WSRFService) – The AnnotationEngine offers the basic annotation management facilities, i.e. create, retrieve, publish, and update. This

service relies on the functionality of the Metadata Management Service (see Section 5.2.2) in order to store and retrieve them as a particular type of metadata associated to an information object. It is worth noting that annotations are created and used in a given context, thus the same information object may have different annotations in different contexts; it is up to this service to identify which annotations are related to a given context.

- **AnnotationPortlet** (Portlet) – This portlet provides the user interface supporting the users in managing annotations. The supported functionality are creation, visualization, editing, removal, listing, and validation, i.e. the comparison with respect to the terms of an annotation ontology.

5.2.4.2 Interactions

AnnotationEngine

The AnnotationEngine interacts with:

- the Metadata Catalog Service in order to store and manage the annotations as particular type of metadata
- the DIS-HLSClient in order to discover the instance of the MetadataCatalog to communicate with

AnnotationPortlet

The AnnotationPortlet interacts with:

- the AnnotationEngine in order to actually execute the operations commanded by the users via the user interface
- the DIS-HLSClient in order to discover the appropriate AnnotationEngine instance

5.2.5 Deployment scenario(s)

Figure 21 depicts a deployment scenario of the services forming the Content & Metadata Management area.

The deployment of the components of the Content Management services is driven by two requirements: performance and availability. We employ a deployment scheme that locates the storage nodes close to data-intense services. The inverse operations of moving code close to the data may seem more appropriate if bulky data volumes need to be relocated. This mechanisms is not implemented by data management services but may be achieved by looking up data management nodes (i.e. their physical location) by the DILIGENT Information Service (see Section 5.1.1) and deploying a consuming service to that (or to some closely located) Grid node. Notice that Content/Storage Management nodes differ from other DILIGENT services as they are coupled with a database instance. That is, relocation of the Content/Storage Management node is, by far, more expensive and time-consuming than other services that do not suffer from this constraint. Detaching Content/Storage Management nodes from their database backend is, in general, a bad idea as (*i*) data first has to be passed to the Content Management service node which routes it to another node that hosts the database (performance) and (*ii*) optimization strategies (through load-adaptive data replication) become much harder to apply as issues like network traffic between Content Management nodes and database nodes (and others) add to the complexity of the optimization problem. To do so, we implement a replication scheme, which has proven to be highly scalable in database cluster scenarios. It comprises an update node that hosts an OLTP database. Any update to the data goes to this node which hosts an always up-to-date full replica of all data that is stored in DILIGENT storage. We will also investigate ways of partitioning the update node into distributed instances to enhance performance and reliability. For instance, a query node may change its role into that of an update node if that fails. Performance evaluations indicate that multiple update

nodes (who need to be coupled through some distributed transaction protocol) often suffer from poor performance due to bad scalability of the transaction protocol.

After the update database has been committed, those update operations are buffered in a global operation queue that regularly broadcasts its entries to query nodes that host a subset of the complete data stock. Each entry holds a local updated queue that is consulted on- demand to conform to (*i*) freshness constraints of incoming queries and (*ii*) the overall data consistency for correct query results.

Partitioning of the data should be conducted in a way that each query execution requires access to one query node only. This is a dedicated Grid requirement, which assumes query nodes to be distributed across autonomous sites driving communication costs into a critical performance bottleneck. In other words, fulfillment of a single task should not impose accesses to many distributed storage management sites. The reason for this is that each accessed site adds "latency" (for establishing the network connection, invoking a web service, etc.) to the overall performance. Exceptions exist for batch-mode operations, where disjoint pieces of data are independently processed (like "batch" Feature Extraction etc.).

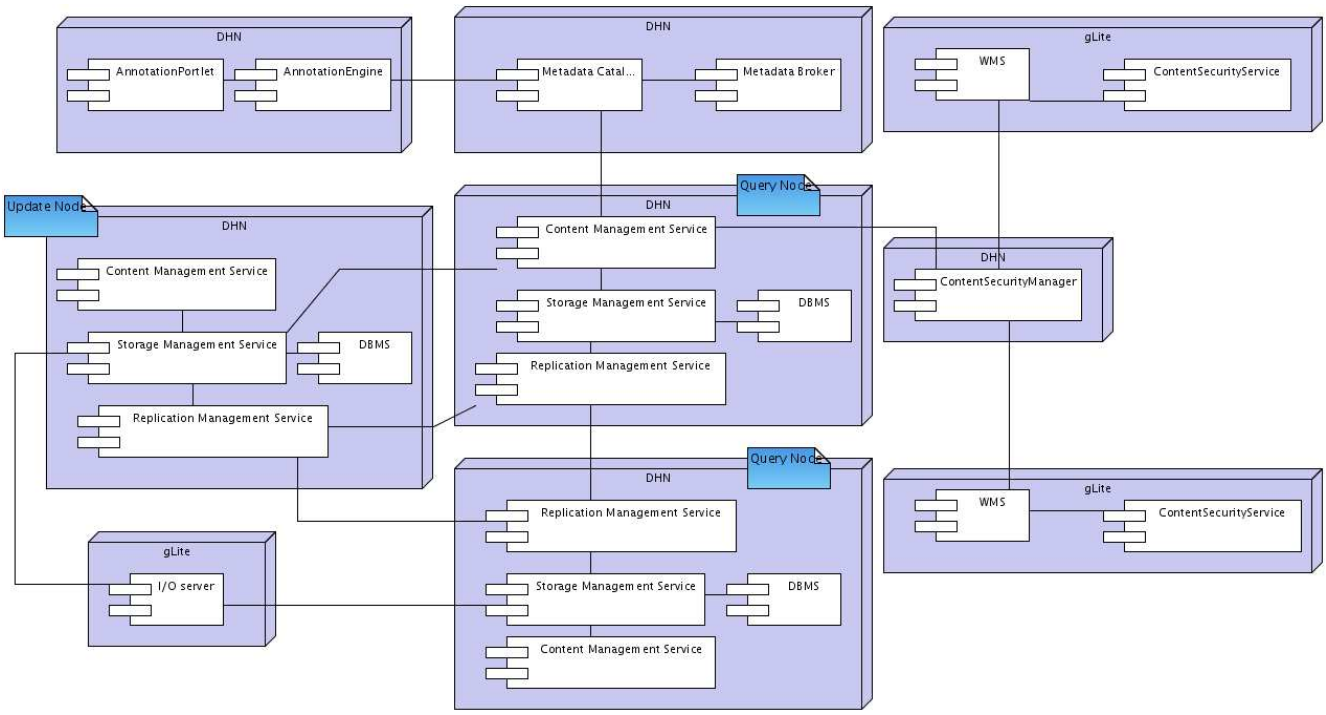


Figure 21. Content & Metadata Management services – deployment scenario

Each storage node (both update and query nodes) is equipped with an instance of the Content Management and Storage Management services. Incoming queries must be transparently propagated to a node holding the queried data. To do so successfully, each node is aware of the data-partitioning scheme, which may be a plain hash-based striping, or a clustering of related information objects, or a self-tuned adaptive partitioning/replication of the data to address the dynamic Grid environment. For instance, collection membership may be considered as a joint clustering criterion. Details of how to maintain the partitioning information will be presented in the upcoming deliverables on the architectural design. The plain scenario of hash-based striping does not require any physical maintenance of partitioning information and has successfully been implemented in cluster database installations. For the time being, the topmost priority is to provide the complete set of Storage/Content Management operations, whereas other aspects of the architecture (like distributed storage nodes) will be introduced subsequently (without changing the Content Management service interface).

Notice that our deployment scheme only links Storage Management components which are responsible of routing requests from Content Management. In fact, different Content Management service instances do not need to be aware of one another.

Updates will always be routed to the update node that must always hold a consistent copy of the complete data stock. The update node may either be a central instance (which permits fast updates but may turn out to be a reliability bottleneck) or a distributed database (which increases reliability through redundancy, but requires scalable distributed transaction protocols). The actual decision depends on the expected update ratio which will be determined by means of experimental evaluations. That is, high update ratio tends to drive distributed transaction protocols into sluggish performance, whereas low update ratios can be coped with by distributed transaction protocols. Also notice that for this purpose we are not considering services that are to be invoked by services in higher levels (such as Content Management).

As for the other services, it is worth noting that they do not have any particular requirement. They can be deployed anywhere (i.e. they only need a generic DHN) in the infrastructure.

5.3 Index and Search services

In this section we provide an architectural overview of the major services developed in the Index and Search area of DILIGENT:

- *Search*, which orchestrates the service invocations and performs the actual lookups during the search operation.
- *Content Source Description and Selection*, dealing with the addressing of content sources capable of satisfying a user query, and Data Fusion, that merges results from heterogeneous content sources.
- *Index Service*, which builds the necessary constructs to efficiently perform searches.
- *Personalisation*, which provides *(i)* the capability to adapt queries and results to the specific end-user preferences and *(ii)* a dedicated user profiles management for query personalization.
- *Feature Extraction*, which processes "raw" data objects to extract special information (features).

Figure 22 and Figure 23 illustrates an integrated architectural view of such services.

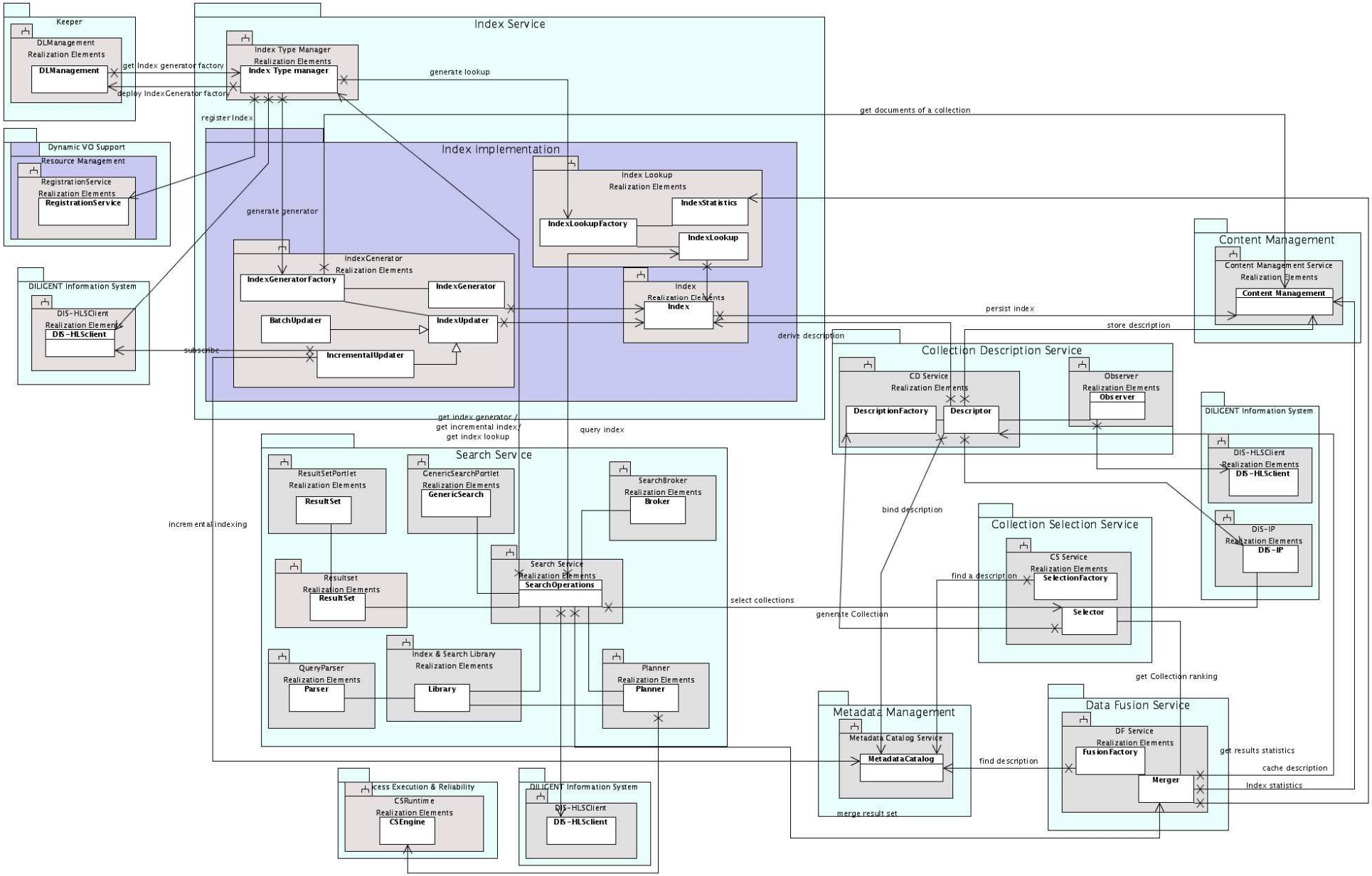


Figure 22. Search and Index services – core architectural view

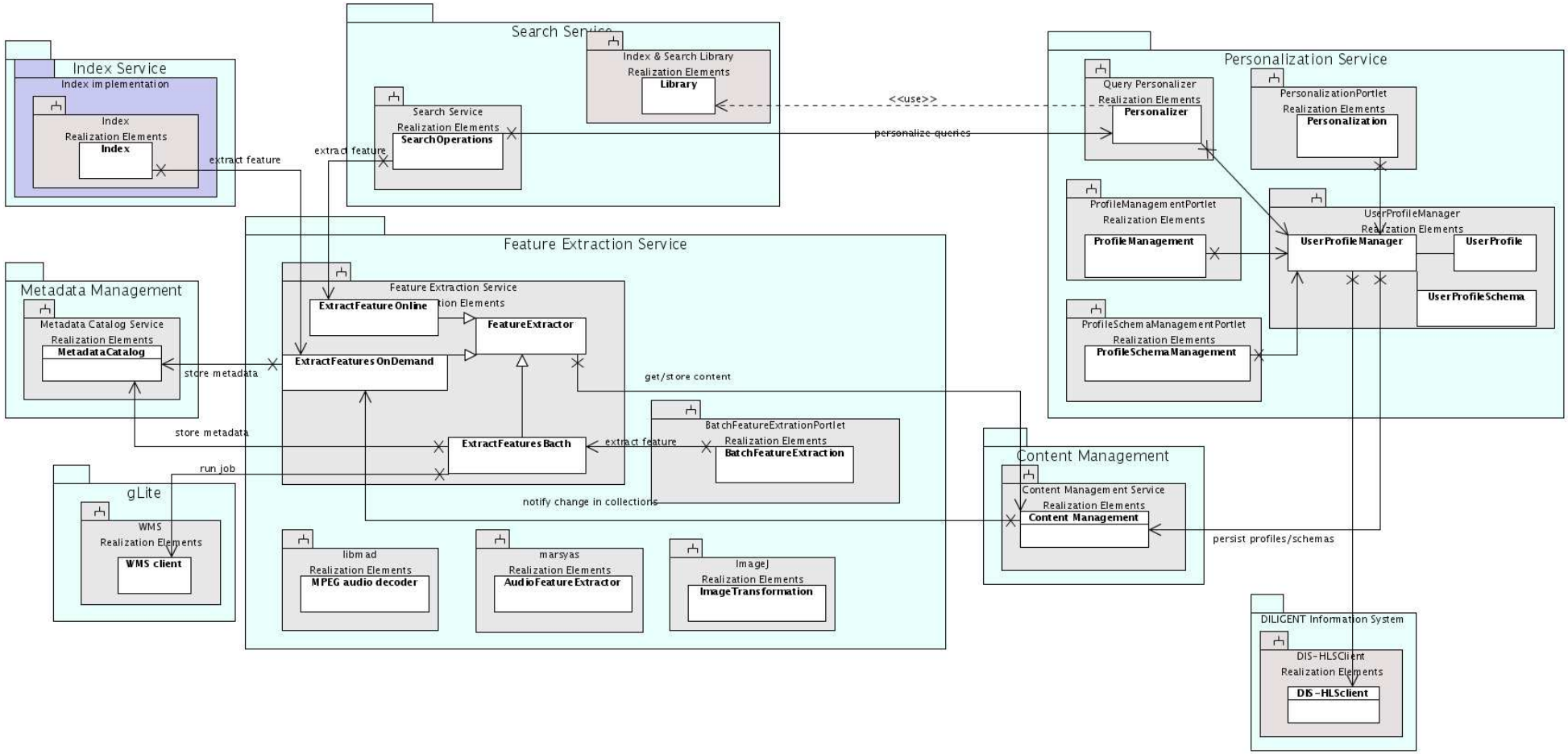


Figure 23. Feature Extraction and Personalisation Services – architectural view

5.3.1 Search Service

The Search service orchestrates the procedure of information retrieval in the DILIGENT environment. The following diagram is a high level abstraction of the operation of the Search in the DILIGENT platform. The diagram offers a rough operational view of the Index and Search group of services and draws the most important dependencies and flows of information inside and outside the Index and Search group of Services.

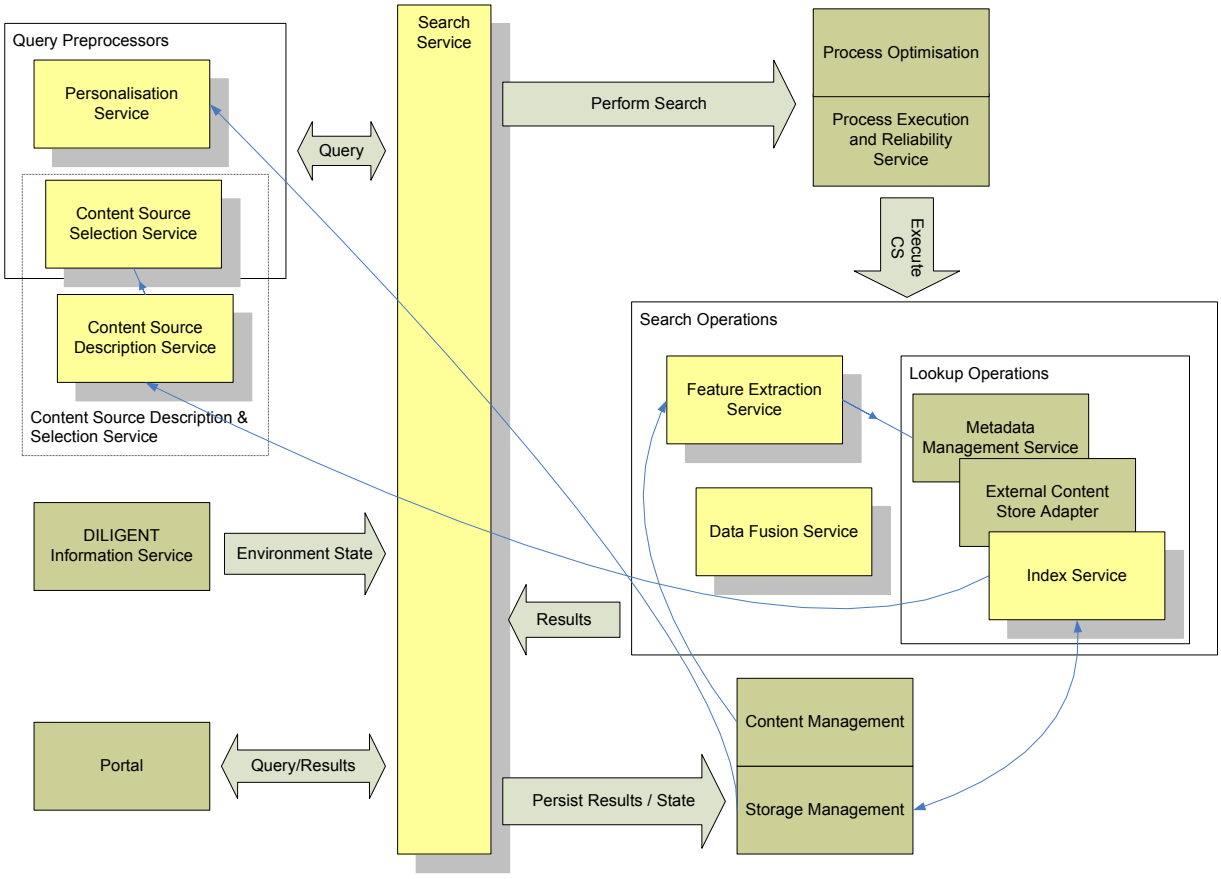


Figure 24. Search Service Operational View

In this diagram elements of the Index and Search group of services are colored in yellow tones while thematically related services are grouped with bounding boxes. The main flows of information (and control) are drawn with heavy arrows while secondary flows are drawn with light ones. Essentially this diagram shows that the Search Service is not directly responsible for providing the actual functionality, rather for redirecting requests to components and services that do the actual work. Yet some of these components are not explicitly referenced as independent services, thus they are implemented as parts of the Search service.

5.3.1.1 Service architecture

The Search Service is composed by:

- **Search Service** (WSRFService) – The Search service is currently considered as a stateless service in its internal operation, since it is executed once and then returns its results to the requester. This is considered to be a two-step serial interaction (even if not implemented as a synchronous operation): Submit -> Return results. The next invocation for a search assumes no session specific state exists. Internally it makes use of stateful services: index service is the most typical example. The results produced are being considered as resource and is treated as a WS-Resource

entity. That is to say, the outcome produced by the execution of the submitted query is handled as a state maintained between subsequent calls to a service coupled with this resource. The above mentioned approach regards "state" as "session state". Yet for performance reasons, one can consider that system state is also handled as service state so as it has to recollect it at subsequent invocations. This would not be an issue even with a stateless service by having system state cached in a cross-session manner. However if this state is multiplexed with user specific attributes, then the need of session state can be justified in order to speed up subsequent service invocations. Because of this an additional issue, still under consideration, is whether or not to treat the information on which Search is based in order to plan its activities (i.e. system status and configuration, along with user-profile-driven state) as stated. The Search service is essentially the factory that creates WS-Resources, e.g. the ResultSets. The adapter pattern is used in order to achieve plug-ability of various query pre-processors or use specific services as Search Operations and Search Wrappers.

- **ResultSet** (WSRFService) –The ResultSet is the service that carries back the results of an operation. Not only it is used to return results to the user but also to transfer intermediate results from one Operation invocation to another by providing a mechanism to optimize the exchange of large chunks of data among various search operators. It is expected to be one of the most common operands for QueryPlan invocations.
- **SearchBroker** (WSRFService) – SearchBroker is an optional service which acts as a front end to a farm of Search Services, taking over the task of dynamically instantiating and invoking the most appropriate ones to serve a user query. The need of its implementation is to be decided yet.
- **QueryPlanner** (WSRFService) – This service is in charge of creating the Query execution plan, i.e. the series of service invocations that near-optimally carry out a search operation.
- **QueryParser** (WSRFService) and QueryParserPortlet (Portlet) – This service (and its portlet) is a tool to enable the user to construct queries object by simple means. Just as for the SearchBroker, it is considered as optional unless stated otherwise.
- **Index & Search Library** (Library) – This library contains the underlying fundamental classes for the various search operations as well as elements to easily interface with the services. A part of its functionality depends and / or extends on functionality existing inside or outside the Search modules.
- **GenericSearchPortlet** (Portlet) –The GenericSearchPortlet is an application-domain independent portlet that allows the most fundamental types of search to be performed in the DILIGENT as well as to present the ResultSets and redirect presenting the documents to the browser capabilities.
- **ResultSetPortlet** (Portlet) – This portlet is in charge of visualizing the ResultSets produced by the search operations.

5.3.1.2 Interactions

Search Service

The Search service is responsible for invoking functionality of underlying services[17] in order to fulfill a request. The most functionally important used services are:

- the Index Type Manager, to generate and update indexes, and to find IndexLookup and IndexStatistics instances

[17] These services are referred to as "Underlying" because they are mostly being "consumed" by the SearchService.

- the IndexLookup, to query existing indexes
- the IndexStatistics, to get overall occurrence statistics of the full content of an index
- the CS Service, to select the appropriate collections given a specific Query
- the FeatureExtractor, to perform an online/on demand feature extraction to match a given Query
- the Personalizer, to personalize the information retrieval behavior
- the DF Service, to merge results that come out from different indexes
- the DIS-IP, to register its own RunningInstances
- the DIS-HLSCLient, to find suitable resources (e.g. identify underlying services and operators which it must consume etc)

5.3.1.3 Deployment scenario(s)

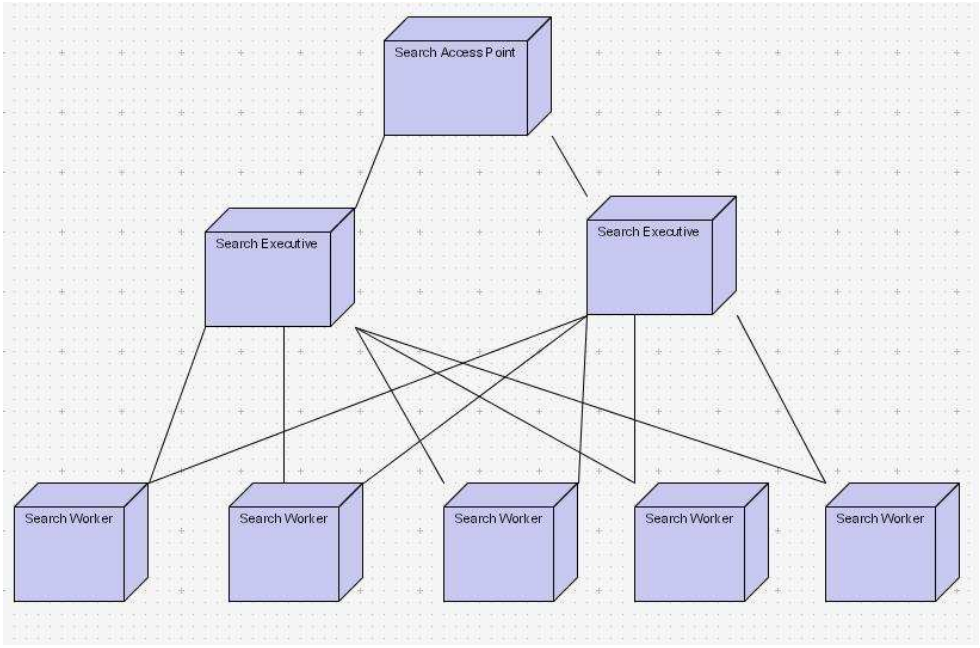


Figure 25. Search services – deployment scenario

This figure describes a typical deployment scenario of search nodes within the scope of one DL. In order to deploy the Search Service, we need one SearchAccessPoint node, one or more SearchExecutive nodes and as many SearchWorker nodes as possible. As an example Figure 25 shows one SearchAccessPoint instance, two SearchExecutive instances and five SearchWorkers.

Generally, we suggest that there should be one Broker Service per DL instantiating and/or invoking functionality of one or more SearchMasters allocated to each DL. Additionally, we suggest to instantiate a large number of SearchWorkers along with QueryPlanners with the latter acting as the workers in the "Master-Worker" pattern of the query execution plan construction. We estimate that at any given point every DHN has an instance of SearchWorker and QueryPlanner services. Since these search components rely on functionality provided by the Index & Search Library package, this has also to be deployed in all these nodes. Finally, for efficiency reasons, we propose that services acting on persistent data be deployed adjacently.

5.3.2 Content Source Description & Selection and Data Fusion Services

The Content Source Description & Selection Service optimizes the execution of content-based queries across a number of collections. It does so prior to query distribution, by

selecting target collections according to the likelihood that their content will prove relevant to the information need underlying the query. Estimations of relevance, in particular, are based on summary descriptions of the content of target collections, or collection descriptions. Independently from optimization issues, the Data Fusion Service supports the distributed execution of content-based queries by integrating the partial results obtained from executing the queries against each of the target collections. To do so, the service *may* rely on collection descriptions. Collection selection and result fusion are orthogonal functionalities, and the latter may well be used without the former.

This section addresses the design of the following services:

i) the *Collection Description Service* (CD): generates, maintains, and provides access to up-to-date sets of statistics about the content of target collections.

ii) the *Collection Selection Service* (CS): matches queries against descriptions of target collections, ranks target collections on the basis of the matching scores of their descriptions, and applies selection criteria to the collection ranking.

iii) the *Data Fusion Service* (DF): merges partial query result lists into a single result list, possibly on the basis of collection descriptions.

5.3.2.1 Services architecture

This group of services is structured in the following components:

- **CD Service** (WSRFService) – This service is responsible for creating, managing, and exposing collection descriptions. A DescriptionFactory F performs the following tasks: (i) it creates WS-Resources for programmatic access to collection descriptions, (ii) it publishes those WS-Resources with the DIS, and (iii) it publishes with the DIS the system-level properties of the deployed instance. For the latter task, F acts as the Web Service front-end of a WS-Resource which has the system-level state of the deployed instances as its stateful component. F is invoked with the identifier IDC of a given collection C and an update policy UP for the description of C which is to be created. F passes IDC and an UpdatePolicy representation of UPD to a DescriptionHome H and receives from H a local identifier IDD to a newly generated Description DC. F uses IDD to qualify the endpoint reference of a Descriptor, where the endpoint reference is specified as part of the configuration of the deployed instance. The qualified endpoint reference QER is to a WS-Resource which has DC as its stateful component and the Descriptor as its Web Service front end. F returns QER to the invoking client for immediate use and publishes it with the DIS for later discovery.
- **CS Service** (WSRFService) – This service is responsible for implementing and exposing collection selection strategies. A SelectionFactory F performs the following tasks: (i) it creates WS-Resources for the application of selection strategies to specific sets of collections, (ii) it publishes those WS-Resources with the DIS, and (iii) it publishes with the DIS the system-level properties of the deployed instance. For the latter task, F acts as the Web Service front-end of a WS-Resource which has the system-level state of the deployed instances as its stateful component. A Selector $SEL_{C1..Cn}$ selects over a set of collections $C_1,C_2,...,C_n$ with respect to a query Q and given selection criteria CR. $SEL_{C1..Cn}$ processes client requests as follows: (i) it extracts the local identifier ID_{SS} of a SelectionSet $SS_{C1..Cn}$ from the qualified endpoint reference used for its own invocation, (ii) it resolves ID_{SS} to $SS_{C1..Cn}$ through a SelectionHome, (iii) it passes Q and $SS_{C1..C}$ to a Ranker R, (iv) it applies CR to the list of RankEntries returned by R, and (v) it returns collection identifiers extracted from the selected RankEnties to invoking clients.
- **DF Service** (WSRFService) – This service is responsible for implementing and exposing data fusion strategies. All the functionality which relates to caching of

collection descriptions is optional, depending on the particular fusion strategy adopted by concrete realizations of the service. A FusionFactory F performs the following tasks: (i) it creates WS-Resources for the application of fusion strategies to specific sets of collections, (ii) it publishes those WS-Resources with the DIS, and (iii) it publishes with the DIS the system-level properties of the deployed instance. For the latter task, F acts as the Web Service front-end of a WS-Resource which has the system-level state of the deployed instances as its stateful component. A Merger $M_{C1..Cn}$ selects over a set of collections $C_1,C_2,...,C_n$ with respect to a query Q and a set of result sets $RS_1,RS_2,...,RS_n$. $M_{C1..Cn}$ processes client requests as follows: (i) it extracts the local identifier ID_{MS} of a MergeSet $MS_{C1..Cn}$ from the qualified endpoint reference used for its own invocation, (ii) it resolves ID_{MS} to $MS_{C1..Cn}$ through a FusionHome, (iii) it applies some strategy to merge the $RS_1,RS_2,...,RS_n$ into a single ResultSet RS, and (iv) it returns RS to invoking clients. Design may vary substantially across particular fusion strategies and is here abstracted over.

- **CD/CS/DF Observer** (WSRFService) – An Observer is associated with each deployed instance of CD/CS/DF service. It is responsible for monitoring external changes which are relevant to descriptions (e.g. indices of bound collections or descriptions of bound collections) and for reflecting those changes onto descriptions.

5.3.2.2 Interactions

CD Service

The CD Service interacts with:

- the Index Service, to derive Collection descriptions
- the Metadata Catalog Service, to publish identifiers of persistent XML serializations of the descriptions of the bound collections as application-level properties of those collections
- the Content Management Service, to persist and retrieve XML serializations of descriptions of the bound collections
- the DIS-IP, to publish instances of (some of) its component services
- the DIS-HLSClient to discover instances of (some of) the component services of the services listed above.

CD/CS/DF Observer

The Observers interact with the DIS-HLSClient to subscribe themselves for notifications about change in Collections and Content Management and Metadata Catalog Services.

CS Service

The CS Service interacts with:

- the Metadata Catalog Service, to retrieve identifiers of persistent XML serializations of descriptions of the bound collections as application-level properties of those collections
- Content Management Service, to resolve identifiers of persistent XML serializations of descriptions of the bound collections
- the Merger, to get Collection ranking
- the CD Service, to register for changes in descriptions of the bound collections and to ask for generating new descriptions
- the DIS-IP, to publish instances of its component services
- the DIS-HLSClient, to discover instances of the (component services of the) services listed above;

DF Service

The DF Service is used by the CS Service and the Search Service. Furthermore, it makes use of:

- the IndexLookup service, to get overall occurrence statistics of the full content of an index
- the Content Management Service, to cache descriptions

5.3.2.3 Deployment scenario(s)

The deployment model of the CD, CS, and DF services does not introduce any further service decomposition over that already presented in the logical view. All the components which conceptually comprise each package will be co-deployed on the same node. Similarly, no service-specific API to distribute across the platform is envisaged at this stage (with the exclusion of stubs to support remote interaction of course).

There are also no constraints to introduce with respect to the co-deployment of services, both with respect to each other and with respect to the other DILIGENT services they rely upon. This simple strategy is motivated as follows.

In principle, a co-deployment strategy is driven by the need to minimize data transfer and storage costs. The need of such optimization is clearly higher for processes which occur in real-time with respect to user interactions. In particular, the optimization of collection selection and data fusion has a higher priority than the optimization of collection description.

Note, however, that the optimization of collection selection and data fusion is currently based on caching collection descriptions in advance of query distribution. Ideally, this caching could be further optimized by co-deploying CS and DF instances and having them sharing the same cache of collection descriptions..

As to collection descriptions, there may be an advantage in co-deploying CD instances and (the look-up part of) Index Management instances. However, the design of the Index Management service shows that the location of the actual index statistics required for collection descriptions may be unrelated to the location of the look-up part of the service. Furthermore, the off-line nature of the CD service and the relatively contained size of collections descriptions with respect to both collections and their indices, indicates that co-deployment requirements have a low priority anyway.

Figure 26 shows baseline design assumptions for the deployment of concrete realization of the CD, CS, and DF services.

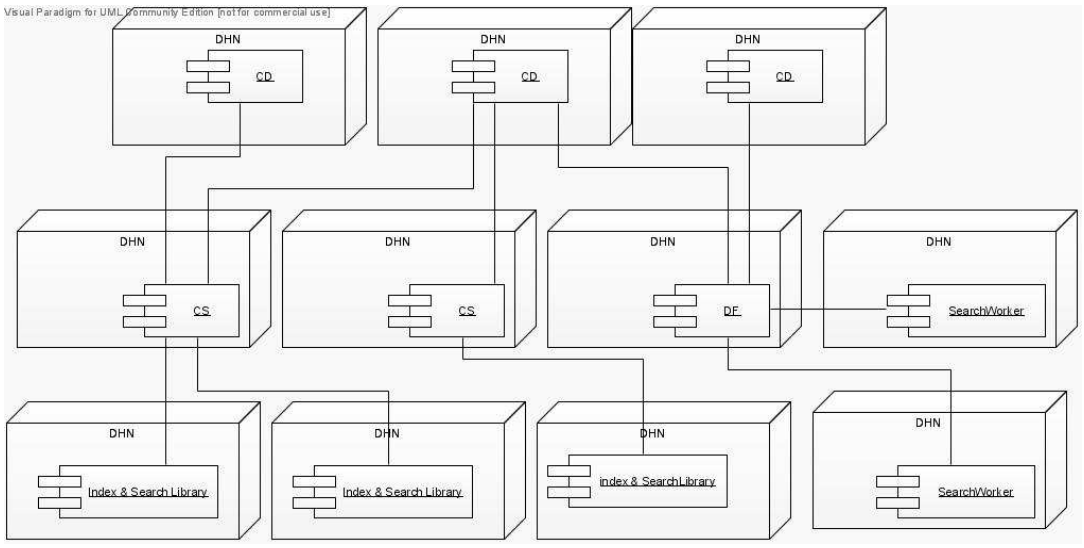


Figure 26 CD, CS, and DF Services – deployment scenario

The diagram shows three CD service instances, two CS service instances, one DF service instance, three Index & Search Library service instances, and two SearchWorker service instances. CS and DF instances may cache and observe descriptions produced by two different CD instances and, viceversa, a single CD instance may produce descriptions used by more than one CS or DF instance. Similarly, a CS instance may support two SearchHelper instances and a DF instance may merge result lists produced by more than one SearchWorker instance. Overall, CD, CS, and DF instances may be relied upon by Search instances which operate in the context of different VDLs.

5.3.3 Index Service

The index service is in charge of producing and maintaining full-text indexes of content in Collections. Indexes are metadata of the Collections, and the index service collaborates with the Metadata Management Service (see Section 5.2.2) in order to associate indexes with collections and maintain index consistency.

The index is considered to be a web resource, identified by a stateful resource identifier, and bound to the index service instance operating on it according to WSRF. There are two ways to implement an index:

- each index is physically represented by a storage object (or group of storage objects), which could be replicated or relocated using the underlying file placement services of the platform. This is supposed to be transparent to the index service, which only operates with logical names for the indexes. For index service implementations built bottom-up for DILIGENT, storage is managed via the Content Management Service (see Section 5.2.1). However, we expect to incorporate legacy (commercial or open source) index implementations, which typically have specific and strict requirements to their underlying storage system. In some cases, they may be able to run acceptably on a regular file system API provided by the grid storage services (and thus profit from the gLite File Transfer and Placement Services);
- some index implementations may need to be statically installed and manage their own storage. For these indexes, the only index "implementation" available as far as other DILIGENT services are concerned is a wrapper that acts as a proxy for the real implementation. (The wrapper will still be located/instantiated via the DILIGENT Collective Services, but there may be restrictions on where and how it can be instantiated, that should be possible to express in the package metadata).

Finally, an index has an index type which determines its properties, as visible for the client, such as the fields, search operations etc. that are available.

5.3.3.1 Service architecture

The service components described here correspond fairly directly to use cases. A major distinction is made between those components providing read-only access to indexes, i.e. index lookup and index statistics, and those services making persistent modifications (creating or updating indexes). The rationale for this is primarily one of authorization and access control: it is simpler and cleaner to give access rights to entire interfaces than to individual methods. Secondarily, concurrency control adds some complexity to the create/update interfaces, and, not least, their implementation. However, there is nothing that prohibits implementations of both sets of services in one and the same package.

- **Index Type Manager** (WSRFService) – The Index Type Manager maintains a repository of index types and provides a starting point for any client of the index service to locate (factories for) the services it needs. The primary functionality of the index type manager is to create, activate or select the appropriate service instances/service implementations to handle an index based on the requested index

type or the index format identified in the index state object. In doing this, it cooperates tightly with the DIS and Keeper services.

- **Service Index Generator Factory** (WSRFService) – When deploying a Package implementing index generation or update functionality, the main entry point to the services provided by the package is an Index Generator Factory. Each factory manages a set of index generator or updater WS-Resources that have been created from it. For purposes of concurrency control, factory methods normally return new, distinct WS-Resources. If concurrent updating is not supported by the implementation, factory methods may fail. The Index Generator Factory is also in charge to associate indexes with collections. It manages the mapping from collection identifier to index (this is probably handled by storing the index identifiers in the collection metadata.). Through methods in this service, clients can directly get references to Index Lookup and Index Statistics WS-Resources for the indexes of a given collection. Finally, It also handles collection-based index generation and update. Once an index has been created, it gets the Incremental Updater for it and subscribes that to the update notifications for the collection. The WS-Resource it creates are:
    - **Index Generator** – The Index Generator WS-Resource is created by an index implementation, i.e. by using the factory method in the index generator factory of the index implementation, in order to generate a new Index in a bounded session.
    - **Index Updater** – The Index Updater WS-Resource is created by an index implementation, i.e. by using the factory method in the index generator factory of the index implementation, in order to generate a new index in a bounded session. Index updaters are either batch or incremental updaters, described in the next two sections. Only commonalities between them are described here. Note that from an external point of view, it is the index lookup and index statistics WS-Resources that act as NotificationProducers, even though the actual changes to the index are produced by the index updater WS-Resources. Notifications are produced when changes propagate to index lookup or index statistics WS-Resource where the NotificationConsumer subscribed. The mechanism used to propagate changes from the index updater to the others is private to the index implementation (although it might use, for instance, the DILIGENT storage layer replication mechanisms), and consequently out of scope for this section. Concurrency control and the level of concurrency permitted depends on the index implementation, both between different updater instances and between updater and lookup/statistics instances.
    - **Incrememental Updater** – The Incremental Updater WS-Resource is created by an index implementation, i.e. by using the factory method in the index generator factory of the index implementation, in order to handle incremental index updates. An incremental updater is normally used as a WS-Notification NotificationConsumer, set up to receive notifications when changes occur, either in the membership of the indexed collection or in individual objects within the collection. In this revision of the specifications, notifications are assumed to be reliable, using the WS-Reliability mechanisms. Guaranteed Delivery is mandatory. Duplicate Elimination is not needed. Guaranteed Message Ordering is optional, and depends on the update profile of this incremental updater. If specified in the update profile, notifications are handled transactionally, i.e. all changes arriving in a single Notify invocation are processed as a unit, and no lookup will see intermediate states.

- **Batch updater** – The Batch updater WS-Resource is created by an index implementation, i.e. by using the factory method in the index generator factory of the index implementation, in order to apply a batch of changes to an index in a bounded session. The batch updater session is terminated and the WS-Resource destroyed either by a Commit or an Abort operation. (All index implementations need not support transactional semantics. This will be requested in the update profile when the getBatchUpdater factory method is called, and the method will fail if not supported.)

- **Service Index Lookup Factory** (WSRFService) – When deploying a DILIGENT package implementing index lookup functionality, the main entry point to the services provided by the package is an Index Lookup Factory. Each factory WS-Resource manages a set of index lookup or statistics WS-Resources that have been created from it. As these services don't modify the index and have no mutual concurrency control issues, the factory methods will normally return an existing WS-Resource if one can be found.
  - **Index Lookup** – The Index Lookup WS-Resource is created by an index implementation, i.e. by using the factory method in the index lookup factory of the index implementation, in order to query the index.
  - **Index Statistics** – The Index Statistics WS-Resource is created by an index implementation, i.e. by using the factory method in the index lookup factory of the index implementation, in order to retrieve aggregate occurrence statistics from the index. Clients interested in updates to the index may subscribe to change notifications on the current version resource property.
- **Index** (WSRFService) – As stated, an Index can be implemented by legacy software (Fast Data Search or Lucene) or by a fully "gridified" software. In the former, the Index is represented by the WSRF service that wraps the legacy software and exposes the Index. In the latter, the Index is a WSRF Service that relies fully on the Content Management Service to manage its persistent storage.

5.3.3.2 Interactions

Index Type Manager

The Index Type Manager interacts with:

- the DIS-HLSClient, to lookup for Index Generator and Index Lookup Factories
- the DL Management, to request the deployment of a particular Index implementation
- the Index Generator Factory, to generate it
- the RegistrationService, to register new Index implementations as DILIGENT Resources (type Service).

Index Generator Factory

The Index Generator Factory interacts with:

- the Content Management Service, to enumerate the constituent documents of a collection, or in case of update, the changes to documents or membership since last update
- the existing indexes, to update them

Index

The Index interacts with:

- the Content Management Service, to persist itself on the underlying grid storage (if it is not implemented using legacy software)

- the Feature Extraction Service (On-Demand Feature Extraction), if it has to index a collection where no features exist

5.3.3.3 Deployment scenario(s)

The figure below depicts a fairly general deployment scenario where index generation and update is initiated by some client on one node, storage for index and the collection being indexed are located on other nodes and search and data fusion is performed on yet another (set of) node(s). The actual index lookup and generation is performed on nodes selected by the index type manager and the generator and lookup factory, and it is suggested that these nodes are the same as the storage for the collection and the index are located upon.

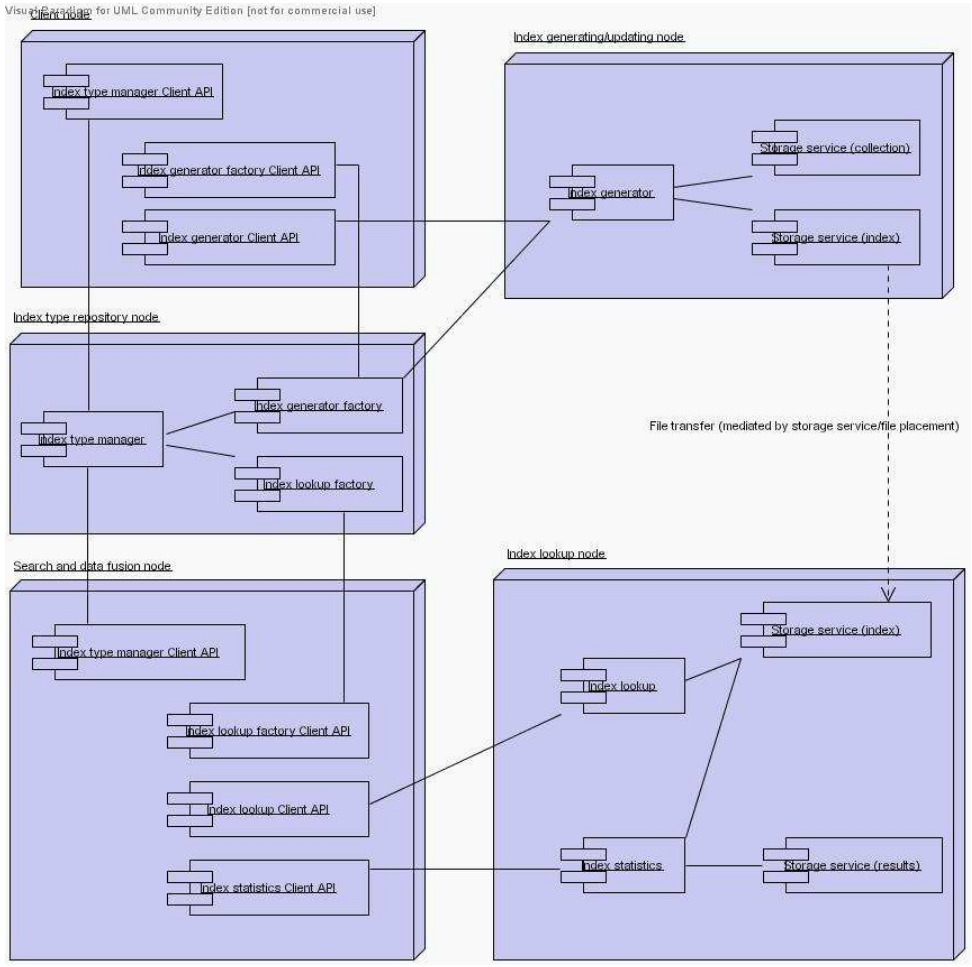


Figure 27. Index Service – deployment scenario

Actual deployments depend strongly on the implementation of the particular index (i.e. index format) and whether the storage service is actually used to manage the persistent store of the index. Two cases spanning most of the possibilities are:

- Index service implemented by legacy software. The software, as well as the DILIGENT Web Service wrappers for it, is statically installed on fixed nodes. Legacy software uses its own persistent storage for the indexes, i.e. outside the control of the DILIGENT content management service. When requested to bind a WS-Resource to operate on a given index, the factory will have to locate one of the existing installations and return a reference to it.
- Fully "gridified". The software components implementing the index service are dynamically deployable using the grid services, and rely fully on the DILIGENT content management service to manage its persistent storage. Factories for such index implementations may select an existing deployment of a particular index, or

deploy a new replica on another node, depending on policy. Replication of indexes is based on the content management service and underlying grid services.

5.3.4 Personalisation Service

Personalization in DILIGENT focuses on the Information Retrieval aspects of the topic, by designing and providing the framework to build highly sophisticated personalisable Digital Libraries. However the reference implementation to be supplied focuses on building the basic blocks of this layer, i.e. does not provide a fully featured service.

Personalization in DILIGENT is handled by using user profiles, i.e. "records" that contain the information necessary for adapting the system behavior on a per-user basis. These profiles are created and maintained by explicit system and user operations. Although creation and maintenance of profiles for DILIGENT are manual, the system is capable of adopting future enhanced implementations that will support automated intelligent profile creation and updating.

The user profiles managed by the Personalisation service are made available to various services for their own use. An example of this could be the Portal Engine and the portlets it hosts, which might (partially or fully) use the DILIGENT user profile as a container for persisting the user interface configuration with regards to a particular user. Such configuration could include the configuration of the front-page portlets, the language, the default collections where user searches are submitted, the user preferences for resource utilization etc.

Guided by the design decisions taken in the Search Service (see Section 5.3.1), information retrieval personalization is not regarded as a special stage of the Search procedure. The case is that personalization service implements a required generic service interface and it is activated and invoked along with other services of the same class, as specified by the Search Service design and implementation.

5.3.4.1 Service architecture

The Personalisation Service has two main groups of functionality:

1. Profile Management that handles profiles, and
2. Query Personalization that personalizes information retrieval.

With regards to this breakdown, the service is composed by:

- **Query Personaliser** (WSRFService) – The Query Personaliser service contains a single class to perform the personalization of the queries. Its only client is the Search service. It adapts a query in order to retrieve information that is most likely to be of interest to the user, either by affecting the retrieved data or by affecting the results presentation. Typical examples of personalization would be:
    - the addition of terms from a set that depicts the user preferences so that that search will flavor the higher ranking of documents that look more interesting to the user
    - preset sorting or presentation options
    - language options
    - filtering of content sources
    - a-priory filtering by specific criteria (e.g. dates etc)
- **UserProfileManager** (WSRFService) – The ProfileManager service is the core component of Personalization service and it is the one that it provides classes to manage and maintain the user profiles. The user profile is expected to be persisted

as an XML document in the DILIGENT storage space[18]. Invocations of the management UserProfileManager service facilities create the following WS-Resource:

- **UserProfile –** This service maintains a WS-Resource for each user profile created. As a consequence it exposes a WS-ResourceProperty for each of these resources. This includes:
  iv. the user identifier the resource refers to
  v. the name of the schema the profile conforms to.
  vi. a logical filename containing the actual structured user profile portions and the links to its unstructured extensions.
- **UserProfileShema** – This service maintains a WS-Resource for each user profile schema created. A UserProfileSchema defines the fields, rules and semantics of the various attributes of the user profile. The service exposes a WS-ResourceProperty for each of these resources. This includes:
  vii. the name of the schema
  viii. a logical filename containing the actual structured user profile schema definition

- **PersonalisationPortlet** (Portlet)– The generic search portlet is an application independent portlet that allows the most fundamental types of search to be performed in the DILIGENT as well as to present the ResultSets and redirect presenting the documents to the browser capabilities.
- **ProfileManagementPortlet** (Portlet) – The Profile Management portlet allows the handling of profile schemas and administration of user profiles.
- **ProfileSchemaManagementPortlet** (Portlet) – The ProfileSchemaManagement portlet allows the handling of profile schemas.

5.3.4.2 Interactions

Query Personalizer

Query Personalizer service mainly depends on:

- UserProfileManager service
- the DIS-HLSClient, for acquiring information related to system configuration
- the Index & Search Library

UserProfileManager

The UserProfileManager and its WS-Resources depend on the Content Management Service for persisting UserProfiles and UserProfileSchemas.

5.3.4.3 Deployment scenario(s)

The current implementation of the Personalisation Service is not computationally intensive. Additionally it is not expected that user profiles require massive storage spaces. However, in order to minimize traffic the profile management should be located next to the storage layer while Query Personaliser should be located next to the Search Engine.

The portlets could either be collocated with the portal engine, or in even better scenario, in dedicated portlet containing servers. The latter decision will be considering if implementation leads to inclusion of a large number of components that are not typically expected to be present in each DILIGENT node. For the time being there is a single

[18] An alternative approach could be to handle user profiles as metadata of user objects. This option is still under consideration but will not be further referenced in this document.

component that is definitely needed for all DILIGENT portlets, which is WSRP4J, yet more might be required in the future.

Due to the above mentioned observations it is expected that Query Personalization service in DILIGENT implementation will be one per DL Instance, will be replicated to sit side-by-side to the Search Service.

Profile Management is also one per DL and replicated as appropriate to minimize traffic and storage misuse.

5.3.5 Feature Extraction Service

The "Feature Extraction" service provides the means to automatically transform the raw content of multimedia documents into features of a different domain. Those features form content-related metadata of a document that is relevant for query processing. For instance, features may be numeric values expressing the color distribution of an image. Features may also be an image's text recognized by OCR.

Within a search scenario, feature extraction comes into play in two places. That is, feature extraction will be initiated on query objects in QBE-like queries at query time. Feature extraction will also be invoked on whole document collections (archives, etc.) to make them "searchable".

Conceptually, we distinguish two general uses of feature extraction according to the output they generate:

1) *Metadata Extraction* is employed to automatically extract metadata from a DILIGENT Information Object. Accordingly, Metadata Extraction's main application is within Search. We have identified three search-related scenarios, where Metadata Extraction is employed:
    - Online Metadata Extraction, i.e. a service that dynamically extracts metadata features from a query object in a QBE-like query (like in content-based multimedia retrieval).
    - Batch Metadata Extraction, i.e. a service that *concurrently* extracts various specified features from documents in a DILIGENT *collection*.
    - On-Demand Metadata Extraction, i.e. a service that is invoked by (i) the indexing service itself (e.g. if indexing service shall index a collection where no features exist; if query processing needs to prune the search space by non-existing features that must be created on-the-fly) or (ii) automatically on notification from storage management on changes in virtual collections (e.g. if an altered membership criteria includes new documents in a collection; if a complete archive is imported).
2) *Data Conversion* is a very generic view onto feature extraction as it transforms a document into another document (possibly of a different domain). This differentiation has become necessary to reflect the concepts of documents (being part of collections which are instances of DLComponents, a DILIGENT Resource) and metadata which is no explicit DILIGENT Resource.

5.3.5.1 Service architecture

The following components are part of the Feature Extraction service:

- **Feature Extraction Service** (WSRFService) – The service extracts features from existing resources and is able to make results accessible as new resources. The service has been designed to operate in a stateless manner: calls to operations create new resources but the service itself does not need to store any information about it's state. This is important to enable scalability since it allows to use an arbitrary number of service instances that do not need to share any state

information and can operate independently. It performs both Metadata Extraction and Data Conversion functionality.

- **ImageJ** (Library) – The library is used to resize images and data conversion.
- **libmad** (Library) – The library is used to decode MPEG audio, e.g. from video content.
- **marsyas** (Library)– The library is used to extract audio feature
- **BatchFeatureExtractionPortlet** (Portlet) – The purpose of this portlet is to provide an interface to managers of collections and administrators to execute batch feature extraction.
- **BatchFeatureExtractionJob(s)** (GridJob) **–** Batch Feature Extraction is designed to take full advantage of the Grid by concurrently extracting different features (like color histograms and texture characteristics on the same images), thus, avoiding duplication in accessing and transporting the data. This set of jobs is executed on the underlying infrastructure to extract such features.

5.3.5.2 Interactions

The Feature Extraction Service depends on:

- the Metadata Catalog Service, to store the output of the Metadata Extractions
- the Content Management Service, to be notified about changes in the content of Collections (On-Demand Feature Extraction)
- the Content Management Service, to store the output of Data Conversion
- the Content Management Service, to retrieve the content of Collections
- the WMS service of the underlying grid infrastructure, to run batch feature extraction jobs

5.3.5.3 Deployment scenario(s)

We propose different deployment views that reflect this distinction in taking advantage of the GRID for those specific scenarios.

Batch Metadata Extraction

Batch Metadata Extraction is applied to a DILIGENT collection which comprises documents of identical media types. It is invoked to make documents in the collection searchable by that feature. It is, thus, likely that *all* (or at least many) available features (like color histogram, texture characteristics, etc.) are simultaneously extracted from the documents in the collection. Therefore, we propose a deployment that:

- installs all feature extractors on grid nodes that are simultaneously invoked on members of a collection and
- replicates this installation onto an arbitrary number of grid nodes to allow for a high degree of parallelization in the batch feature extraction process.

First requirement pays out in avoiding duplicate access and transport of documents from DILIGENT storage. Second requirement permits a disjoint partitioning of the workload taking into account the current task load of each participating node.

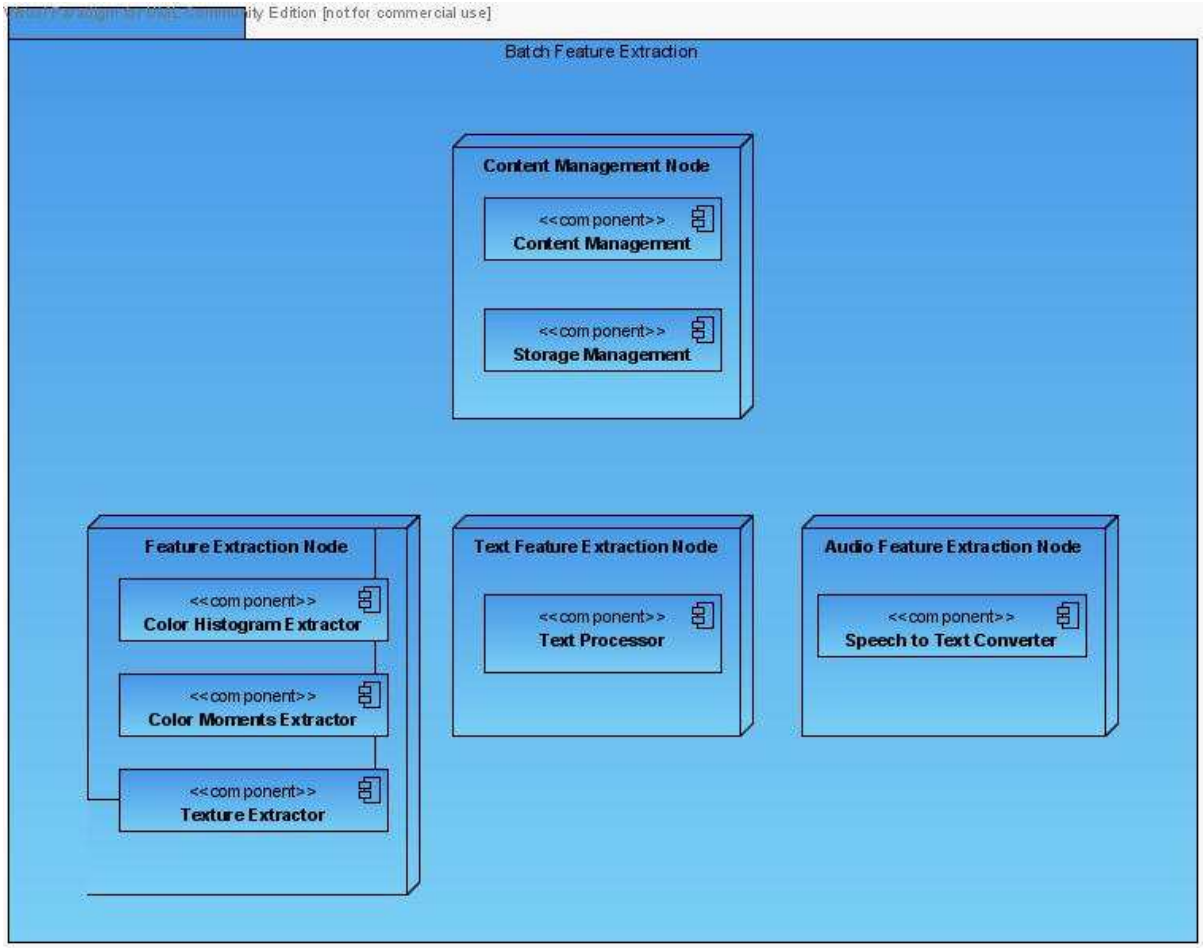


Figure 28. Batch Metadata Extraction – deployment scenario

On-Demand Metadata Extraction

In contrast to Batch Metadata Extraction, On-Demand Metadata Extraction is invoked on smaller numbers of documents, not necessarily forming collections. Moreover, its execution is tightly coherent with other services, namely changing of a collections membership criteria, search, archive import, and other updates to documents stored in DILIGENT storage.

We propose a deployment of On-Demand Metadata Extractor services close to the physical storage location of affected documents to avoid performance bottlenecks in transporting the data across the network which might become critical since:

- Updates to documents may happen fairly often, even if changes may be small. From a performance point of view, it is dissatisfactory to send an updated document across the GRID each time since I/O times may easily exceed computational efforts for the actual feature extraction.
- Query processing (i.e. search) may decide to incorporate additional features on a subset of searched documents (e.g. to decrease the number of documents in a result set based on user interaction; to prune the search space when other features fail to do so efficiently). In this case, overall execution time for feature extraction becomes critical to permit an acceptable (i.e. interactive) behavior of the search service.
- When archives (like from 3rd party data sources) are imported into DILIGENT storage, those documents should not be transported to two locations (for actual storage and for feature extraction) which tremendously increases network load at

that time. In contrast, transporting documents for batch feature extraction is acceptable since advantages of distributed parallel feature extraction outweigh the efforts for data transportation.

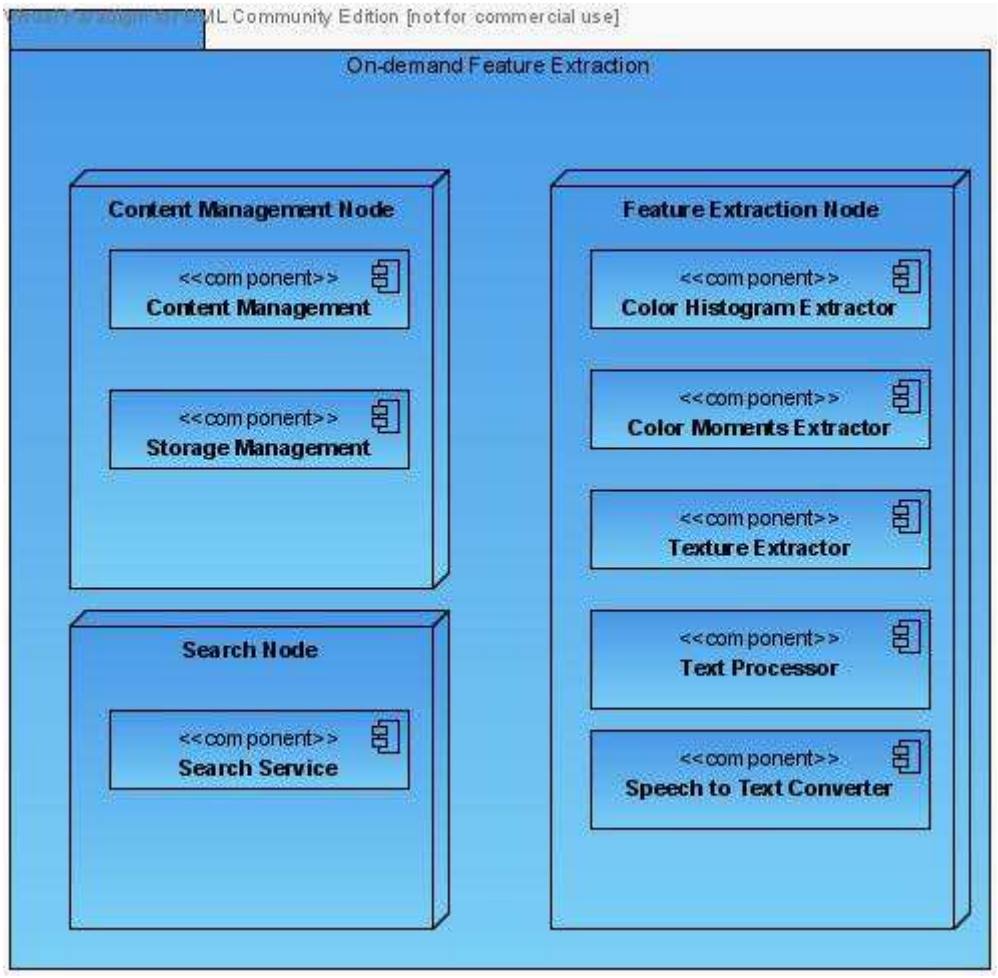


Figure 29. On-Demand Metadata Extraction – deployment scenario

Online Metadata Extraction

Online Metadata Extraction is the most time-critical services of feature extraction. On the one hand, it is invoked in any QBE-like search that passes (one or many) documents as query times to the search service. On the other hand, it is completely independent of storage since extracted features are transient and are not stored permanently (neither of the query documents themselves).

Complex queries encompass (1) several query objects referring to (2) many (different) features ("multi-object multi-feature queries"). These characteristics can be exploited in order to allow for speed-up through parallelization of Online Metadata Extraction. We propose a deployment of feature extractors close to the search service and/or the persistent storage of the index. Since an index is built on top of certain feature type(s), this information should be taken advantage of to choose the feature extractors that are deployed close to a specific index. Parallelization comes into play if several nodes are grouped around an index where each node is responsible for (1) one query document or (2) one employed feature type.

5.4 Process Management services

Under the name *Process Management* is grouped the set of services responsible for the creation, validation, optimization and execution of compound services (CS), a.k.a.

processes[19]. A compound service is a composition of existing (atomic or themselves compound) services, combined together in a workflow. In the following, we distinguish between:

- the CS definition (or specification), that defines the workflow; it is a particular kind of DILIGENT Resource, as introduced in Section 3.1.1, and
- the CS execution, that is the phase in which a defined CS is executed by some Process Management components

Process Management has been organized into 5 services, each responsible for a particular aspect:

- *Process Design & Verification*, responsible for providing a user interface for viewing, editing and managing process definitions and for validating CSs defined by a user or generated by another DILIGENT service or the DILIGENT system.
- *Process Execution & Reliability*, which includes: (i) UI components enabling the user to control the execution of CSs (run, abort, and monitor), and (ii) a component running on each DHN, responsible for the actual execution of the process.
- *Process Optimisation*, responsible for optimizing the workflow defined in a CS specification by transforming or modifying it into an optimized workflow.
- *Process Resources System* serving as a partial abstraction layer tailored to the needs of the process management services, easing access to the existing resource-related functionality provided by the DILIGENT Information System and Dynamic VO Support.
- *gLite Job Wrapper*, providing a service-oriented access to gLite job functionality. By the means of this service, existing gLite jobs can seamlessly be included in compound services – or be called independently through a web service.

Figure 30 depicts the components forming these areas and an integrated picture of the Process Management services. The diagram also highlights the "connectors" that link the components together. The following sections briefly introduce each component and describe its main interactions. A more complete description of Process Management services can be found in [20].

[19] The terms "Compound service" and "process" are used as synonyms throughout this section.

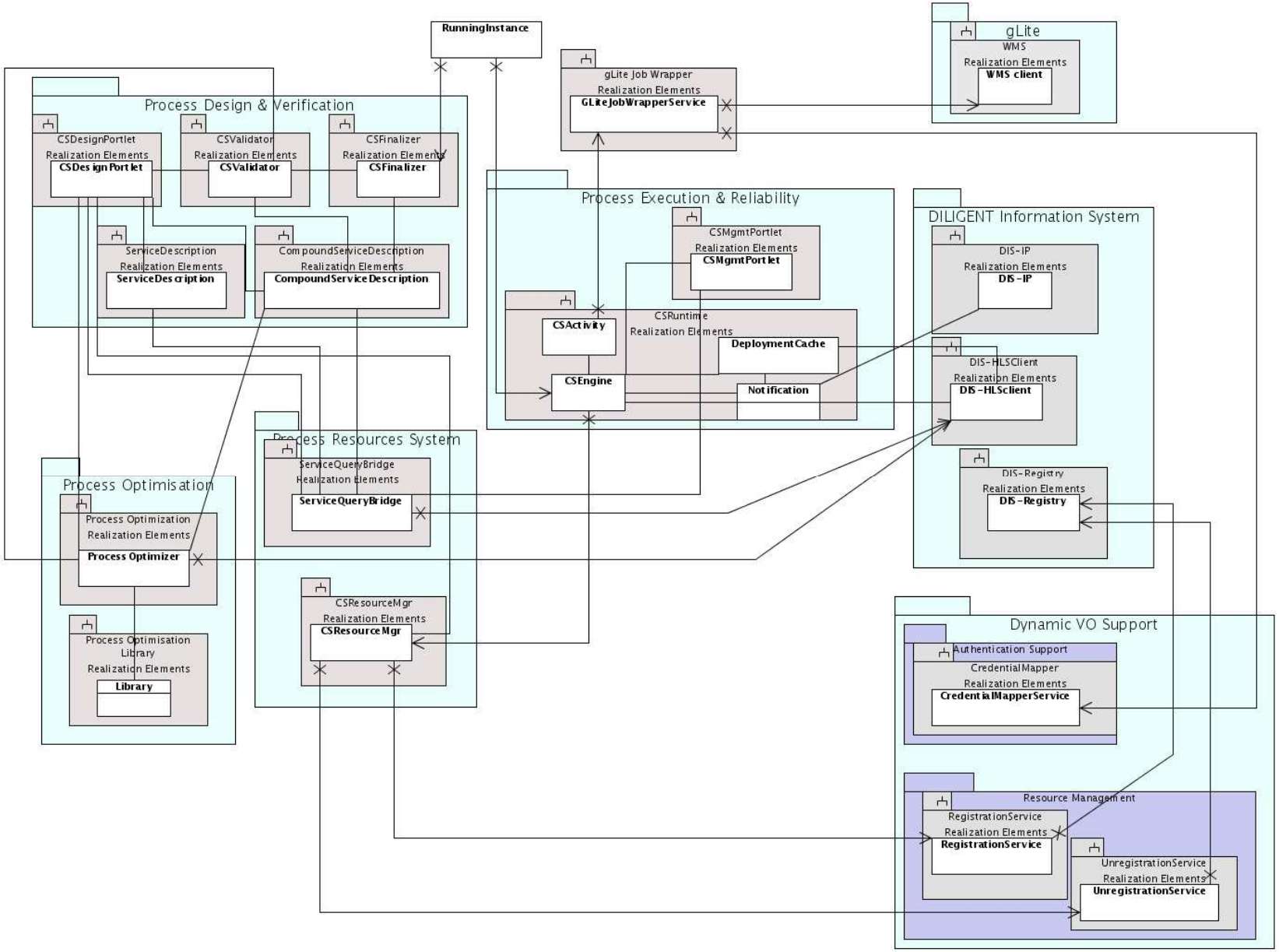


Figure 30. Process Management services – architectural view

5.4.1 Process Design & Verification Service

The Process Design & Verification provides all functions related to Compound Service specifications. These functionalities are provided to the user by means of a user interface implemented via a portlet that is hosted by the DILIGENT portal. Before a process can be actually run, it has to be validated. Validation (or verification[20]) checks for the correctness of the process specification, and only if the specification is correct, marks the process definition as being correct. The validation of processes is done in another component of the Process Design & Verification area, but it can be invoked through the portlet UI.

5.4.1.1 Service architecture

Five components implement the Process Design & Verification functionality:

- **CSDesignPortlet** (Portlet) – This portlet enables the user interface to perform the following operations: *create*, *load*, *edit*, *explore*, and *remove* CS specification as well as *explore service description*. It also gives the possibility to perform *validation*, *optimisation*, and *saving* CS specifications, by presenting appropriate user interfaces to the user.
- **CSValidator** (WSRFService) – This service performs syntax checks, semantic checks, and, any transactional or other additional properties the process specification may have. The verification has to ensure that different types of constraints are met by the specification.
- **CSFinalizer** (WSRFService) – This service provides non-interactive access to functions for *validating*, *optimizing* and *saving* compound service specifications created by other DILIGENT services, e.g. because a customized process created on-the-fly needs to be invoked.
- **ServiceDescription** (Library) – This library provides the abstract classes needed to represent a service description. It contains classes needed to represent the attributes of both atomic and compound services.
- **CompoundServiceDescription** (Library) – This library provides the abstract classes needed to represent a compound service description. It contains classes and facilities enriching those of the ServiceDescription pertaining to the specification of a Compound Service.

5.4.1.2 Interactions

CSDesignPortlet

The CSDesignPortlet interacts with:

- the CSValidator, to perform the CS check activity required by the user
- the ProcessOptimizer, to perform the CS optimization activity required by the user
- the ServiceQueryBridge, to indirectly query the Information Service in order to acquire information about the services and be thus able to assist the user during the CS specification phase
- the CSResourceMgr, to support the creation and deletion of CS specifications

CSValidator

No interaction is envisaged since the service performs internally all the checks.

CSFinalizer

The CSFinalizer interacts with:

- the CSValidator to perform the validation of the CS

[20] Validation and verification of CS are used as synonyms throughout this section, from now on.

- the ProcessOptimizer, to perform the CS optimization activity
- the CSResourceMgr, to support the creation of CS specifications

5.4.2 Process Execution & Reliability Service

5.4.2.1 Service architecture

Five components implement the Process Execution & Reliability Service:

- **CSMgmtPortlet** (Portlet) – This portlet provides the user with the possibility to perform *selecting CSs*, *starting a CS*, *monitoring CS execution*, *aborting the execution* of a process, *removing the state* information of a finished process.
- **CSEngine** (WSRFService) – This service provides the operations relevant to process execution, such as providing the methods for starting/finishing process execution, maintaining (global) process state, handling splits and joins in process control flow, handling failures, routing and handing over process execution to the successor activities, etc.
- **DeploymentCache** (Library) – This Library represents the cache; it accumulates notifications about the services being watched and provides this information to other classes querying the cache. The cache is populated by subscribing (using the Notification) to DIS change messages concerning the services/service instances/DHNs to be watched; the information on which services actually need to be watched is obtained from the CSEngine.
- **CSActivity** (Library) – This library is responsible for executing the call of the service instance defined as the "target" of the current process activity. This will in most cases be a DILIGENT Running Instance hosted on the same DHN node as the CS Runtime. However, it may also be a non-DILIGENT web service running on an external machine.
- **Notification** (WSRFService) **–** This service contains the logic for receiving and sending notifications from/to the DIS and is used by many of the other classes of this service.

5.4.2.2 Interactions

CSMgmtPortlet

The CSMgmtPortlets interacts with:

- the ServiceQueryBridge, in order to identify the CS to be executed or to discover a CS instance for its management. This interaction actually allows having access to the CS profiles stored on the DIS Registry as well as the CS instance stored on the DIS.
- the CSEngine, in order to perform (i) the execution of a selected CS specification, (ii) the monitoring and the aborting of an identified CS instance, and (iii) the manipulation of the information about a finished CS instance. This third point is needed since logging and state information are kept locally at the executing nodes, and since service execution is asynchronous (the execution is decoupled from the control in the sense that a user does not have to wait until a process terminated), the results of a process execution and its logging information may have to be kept until retrieved by the user (or possibly until they are automatically purged by the system after some timeout).

CSEngine

The CSEngine interacts with:

- the DeploymentCache, in order to identify the proper successor node, i.e. the DHN hosting the running instance of the service to be executed next within the CS. Moreover, thanks to this interaction the CSEngine has access to information

- the Notification, in order to manage the request about services to be watched. The resulting call-back messages are queried by interacting with the DeploymentCache.
- the ResourceMgr, in order to register/unregister the CS instance it is running.
- the DIS-HLSclient, in order to publish information about the global CS status.

5.4.2.3 Deployment scenario(s)

The CSRuntime must be deployed on each DHN of the infrastructure where the CSs can be executed.

5.4.3 Process Optimisation Service

The Process Optimisation Service (POS) defines a framework for optimizing workflows of operations[21]. The term "optimisation", with respect to the context, refers to the quest for a manner (execution plan) to obtain the expected result while complying with some external, non-functional constraints. These non-functional constraints are usually expressed with a cost estimation function, which makes optimisation to become an operation to find an execution plan that maps to the lowest cost value.

5.4.3.1 Service architecture

Two components implement the Process Optimisation Service:

- **ProcessOptimiser** (WSRFService) – This service accepts a workflow and processes it according to service description presented here and the specific algorithms to be adopted. This operation is guided by a fundamental rule: the produced plan must have the same results as the original workflow and be capable of direct execution through the CS Engine. The ProcessOptimiser is responsible for the orchestration of the optimisation process. It parses the workflow, invokes CSValidator and obtains all available information on resources in order to be able to calculate the plan cost during subsequent steps. After collecting this information it spans one or more Planners (processes) to search for a suitable execution plan. During this operation it acts as a master and controls the worker operations and lifetime[22]. The fittest plan is returned to the requestor.
- **ProcessOptimizerLibrary** (Library) – The library provides a set of helper classes supporting the process optimization phase. In particular, it includes classes to represent a process as well as a set of classes enabling to calculate or estimate the cost of a service in a process.

5.4.3.2 Interactions

ProcessOptimiser

The ProcessOptimiser interacts with:

- the CSValidator service, in order to validate both the submitted workflows as well as the optimized plans generated
- the DIS-HLSclient, in order to gather the needed information about the resources constituting the workflow to perform the optimization process

[21] Initially optimisation has been referenced as consisting of two different operations: one hosted in the Search Service, aiming at query optimisation and one hosted by the Process Management Services, aiming at CS optimisation. However, latest decisions in the Search Service design process, which now targets CS Management as the one to carry out the execution of a query, lead to reconsideration of our initial planning, offering more opportunities of reuse and a more robust design.
[22] Parallelisation under the Master-Worker approach will be considered if suggested by performance measurements.

ProcessOptimiserLibrary

To acquire the information needed to the cost calculation/estimation algorithms, the Process OptimiserLibrary interacts with:

- the single services
- the DIS-HLSClient

5.4.4 Process Resources System

The Process Resources System is introduced here because *(i)* it provides functionality which cannot be unambiguously assigned to any of the preceding services and *(ii)* it serves as a *partial* abstraction layer to some of the information provided by the DIS, tailored to the needs and functionality of the process management, especially for higher-level searches for service *descriptions* (i.e. profiles, see Section 3.1.2.2) within the DIS. As such, it is not meant to completely "shield" the DIS from the components of the other services, but rather to complement it, providing commonly needed functionality through a (potentially) simpler interface than the DIS provides.

5.4.4.1 Service architecture

Two components implement the Process Design & Verification Service:

- **ServiceQueryBridge** (WSRFService) – This service is the point of reference for queries pertaining to (compound and atomic) service descriptions. It provides an internal interface allowing for easy querying of service specifications; the queries are translated and forwarded to the DIS-HLSclient.
- **CSResourceMgr** (WSRFService) – This service is responsible for the creation and deletion of process specifications and instances. In particular, the former is stored on the RegistrationService as new DILIGENT Resources while the latter are exposed to the local DIS-IP as WS-Resources.

5.4.4.2 Interactions

ServiceQueryBridge

The ServiceQueryBridge interacts with the DIS-HLSclient in order to provide Process Management with an internal interface allowing to easily query (compound as well as atomic) service specifications stored on the Information System.

CSResourceMgr

The CSResourceMgr interacts with:

- RegistrationService of the Dynamic VO Support, in order to register a new CS specification
- the Unregistration Service of the Dynamic VO Support, in order to remove a CS specification among those available in DILIGENT as resources.
- the DIS-IP, in order to register the CS instances

5.4.5 gLite Job Wrapper Service

The Job execution possibilities of gLite have to be integrated and be seamlessly accessible from within the DILIGENT infrastructure. However, since it is not feasible to completely combine gLite's job execution and the SOA-oriented approach of DILIGENT – in the sense of "jobs composed of (compound) services and vice-versa", we decided to provide the combination possibility in one direction: users will be able to integrate gLite jobs into compound services, just like "ordinary" services. This service provides the means to execute gLite jobs transparently by calling a DILIGENT service.

5.4.5.1 Service architecture

The GliteJobWrapperService (a WSRFService) implements the gLite Job Wrapper Service. This service is in charge of interfacing with the services of the gLite infrastructure in charge of providing the job submission facility, namely the Workload Management System (WMS) and Logging and Bookkeeping (LB).

5.4.5.2 Interactions

The gLite Job Wrapper interacts with:

- the CredentialMapper, in order to obtain a certificate recognized and usable as a valid identity to submit jobs to the gLite infrastructure
- the gLite WMS, in order to submit jobs to the gLite infrastructure
- the gLite LB, in order to monitor the status of the submitted jobs

5.4.6 Deployment scenario(s)

Figure 31 depicts a deployment scenario for the services and components forming the Process Management Service. In particular, we decided to deploy the portlets providing the interfaces for CS design and verification (CSDesignPortlet) and CS execution and reliability (CSMgmtPortlet) on the same hosting node even if they can be deployed on different nodes. The same holds with respect to CSValidator and CSFinalizer, they can be deployed on different nodes. With respect to the execution of process, we have four hosting nodes equipped with the components of the runtime environment, and thus the execution of the process will be distributed among them according to the process specification. The final mention is about the process optimization; in our scenario we decided to have only one of those services, even if this type of service can be replicated because each instance is able to work independently.

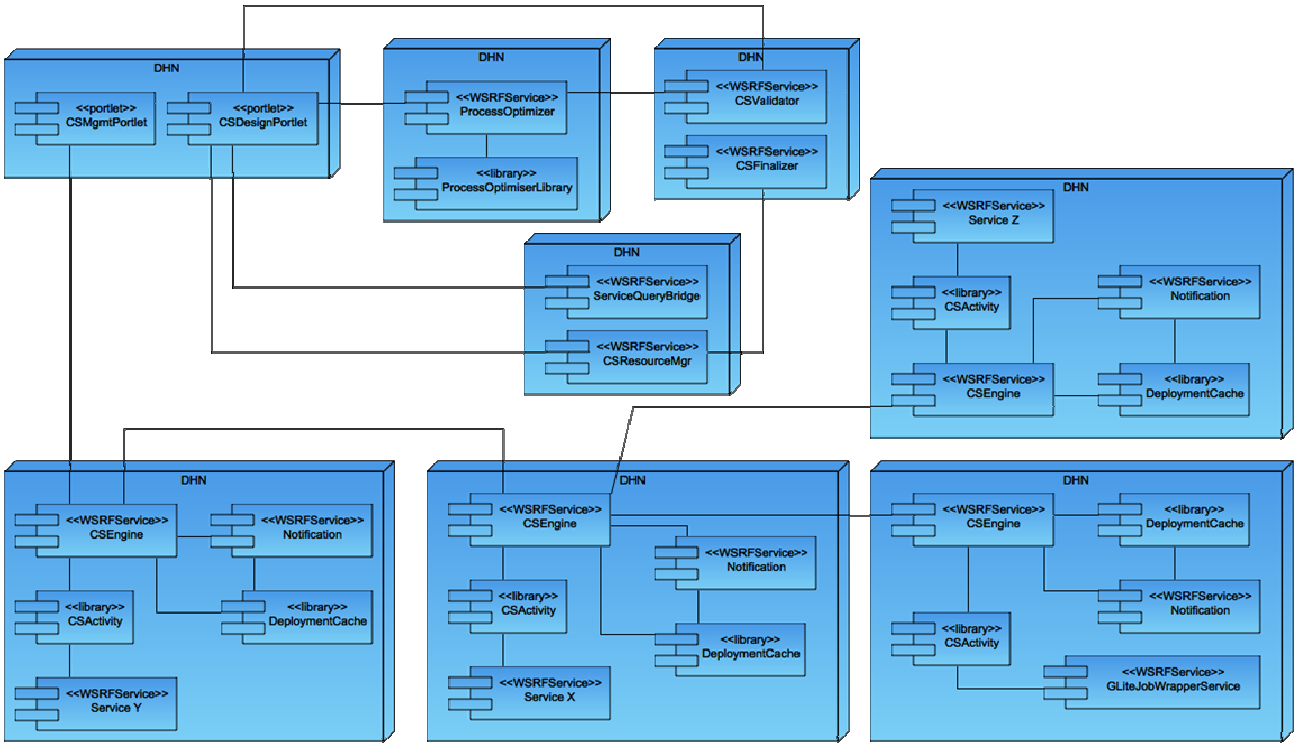


Figure 31. Process Management Services – deployment scenario

5.5 User-Community Specific Application Services

The goal of the services forming this functional area is twofold:

1. to provide the user communities with the user interface and the related technologies enabling them to have access to the DILIGENT functionality, and
2. to provide community specific services fulfilling the particular needs of a real world application scenario.

As stated in the brief descriptions of the components presented in the previous sections, the user interfaces rely on the portlet technology [12]. This technology is presented in Section **Erreur ! Source du renvoi introuvable.**.

The portlets are mainly organized in two groups:

- the back-end portlets making interfaces to perform administrative tasks (such as the K-UI portlet, see Section 5.1.3.1).
- the portlets providing the user interface with the end-user applications

While the latter has been described in the previous sections when presenting the architecture of each service, the former is strictly related to the two complementary application scenarios, the ARTE scenario and the ImpECt scenario, that DILIGENT supports.

The community underlying each scenario expressed a set of requirements reporting the needs for particular objects and interaction patterns that are provided by relying on the basic DILIGENT functionality and services presented until now via appropriate portlet. In particular:

- the ARTE community express the needs for managing courses, workshops and exhibitions
- the ImpECt community asks for functionality dealing with the management of reports. Reports are digital objects dynamically generated in accordance to a defined template. The logic underlying these objects relies on top of the Content & Metadata Management Service (see Section 5.2) and Process Management services (see Section 5.4).

Further details about the functionality requested by the DILIGENT end-user communities are reported in [21] and [1].

As it can be identified in these documents, there is a set of functionalities requested that does not fall within the scope of a Digital Library Management system such as DILIGENT. Additionally there exist well-placed open-source systems and well-defined open standards that facilitate or provide these sets of functionalities in the world of legacy applications. However porting these to the grid (i.e. rendering them gLite based) would require a whole independent project. Yet it is the desire of the DILIGENT members to satisfy as much as possible of these functionalities as part of the initial user scenarios to be instantiated by the platform.

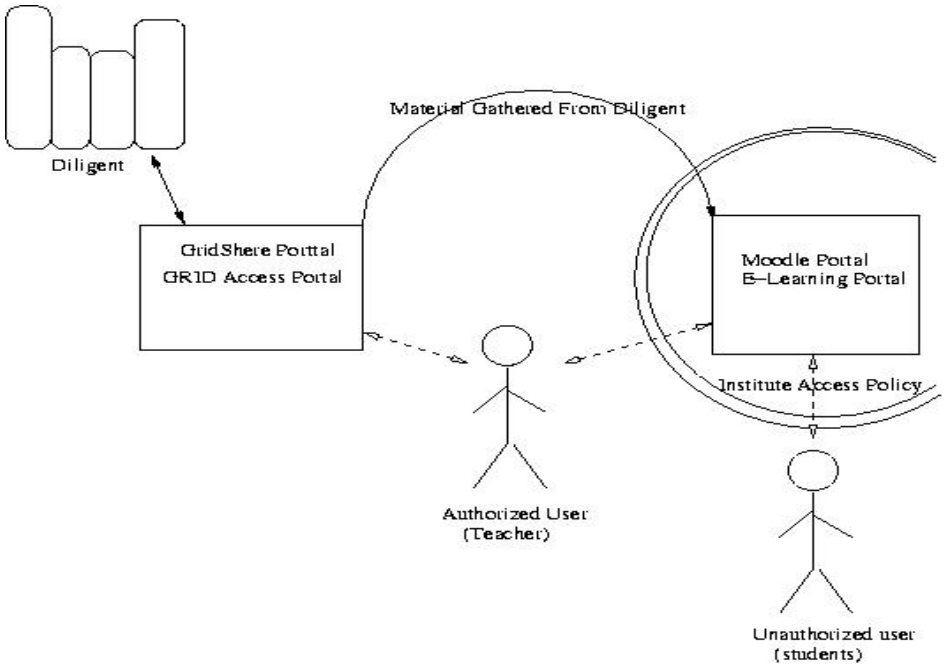
The approach to address this will be towards deploying legacy applications on the boundary of the infrastructure and bridging them (if required) to the DILIGENT infrastructure so that they can exploit it

Enabling the e-learning society to harvest the benefits of using the GRID and more precisely to facilitate the services provided by GRID enabled Libraries is a case of the afore-mentioned bridging. More specifically, facilities provided by the well-known e-Learning Portal Moodle will be bridged with the DILIGENT platform. In this scenario with regards to permissions there are two types of users they will have to access the Diligent infrastructure.

1. Instructors who will be authorized to access the digital libraries stored in the GRID.
2. Students who will not be able to access the GRID (unless they own the appropriate credentials).

Authorized personnel will be able to search the GRID and the DILIGENT infrastructure so as to retrieve relevant material. This personnel which will actually be the instructors will own certificates and use them in their quest for material to include in courses. The functionality of accessing the libraries of DILIGENT, Searching and Browsing the results is covered by the general DILIGENT Portal. The material gathered will have be to presented to the students or to any other interested through Moodle. It will also have to be shaped in the form the course author/instructor wants it to be. This includes annotating the material creating HTML texts that connect to each other and include links towards the material gathered from the GRID. The authorization and authentication process of accessing the e-Learning Portal could take place in various ways according to the institutions policy.

As shown below the instructor will be the one in the middle of two front-ends that has to port any desired material from the DILIGENT to the e-learning environment. The material will be packed in a standard form (SCORM) so that openness is achieved and alternative options of e-learning systems are not ruled out. Primitive forms (plain files) of the material to be transferred will be also provided.



In addition to that, more open-source systems such as end-user forums, discussion boards, etc will be laid at the boundary of the infrastructure to offer to the user communities a rich set of facilities.

Finally, even if not explicit mentioned by the communities, another important aspect dealing with application scenario is related to the implementation of appropriate and specialized wrappers enabling the Content Management Service to import external collections, archives and information sources into the DILIGENT system.

The Content Management is equipped with general-purpose components in charge of implementing this functionality but there could be the need to customize and/or develop appropriate wrappers for certain content sources.

5.5.1 DILIGENT Portal

A *portal* is a Web-based desktop that is customizable both in the look and feel and in the content and applications which it contains. A portal, furthermore, is an aggregator of content and applications or a single point of entry to a user's set of tools and applications.

GridSphere (www.GridSphere.org) is an open-source framework to be considered as the primary candidate platform for portlet-hosting[23]. It enables developers to develop and package third-party portlet web applications that can be run and administered within the GridSphere portlet container. For its purposes, it uses the following standards:

- JSR 168 Portlet API, to provide reusable web applications
- Java Server Faces, to specify an event based user interface for web presentation development

Moreover use of WSRP (Web Services for Remote Portlets) standard [30] Is assumed as the most appropriate technology, in order to deploy our portlets under the SOA paradigm. The WSRP standard specifies how compliant portals can be consumed as web services. However since GridSphere does not provide support for WSRP portlets out of the box its integration to the DILIGENT platform is expected to be performed in the near future.

[23] Although GridSphere is assumed to be the portlet-hosting platform, other alternative options exist e.g. uPortal (www.uportal.org).

The WSRP specification defines how to leverage SOAP-based Web services that generate mark-up fragments within a portal application. By defining a set of common interfaces, WSRP allows portals to display remotely running portlets inside their pages without requiring any additional programming by the portal developers. To the end-user, it appears that the portlet is running locally within their portal, but in reality the portlet resides in a remotely running portlet container, and interaction occurs through the exchange of SOAP messages. By using WSRP within Service-Oriented Architecture, presentation-oriented portlet applications can be discovered and reused without engaging in additional development or deployment activities.

Erreur ! Source du renvoi introuvable. illustrates WSRP portal architecture and the role a portal plays in aggregating the mark-up fragments. Although this diagram shows a portal consuming WSRP portlets from only a single producer, there is no reason why a portal could not consume portlets from any number of WSRP producers. The WSRP Specification defines the following actors within a WSRP architecture:

- **WSRP producer:** This is a Web service that offers one or more portlets and implements a set of WSRP interfaces, thus providing a common set of operations for consumers. Depending on the implementation, a producer could offer just one portlet, or could provide a run-time (or a container) for deploying and managing several portlets. The WSRP producer is a true Web service, complete with a WSDL and a set of endpoints.
- **WSRP portlet:** A WSRP portlet is a pluggable user interface component that lives inside a WSRP producer and is accessed remotely through the interface defined by that producer. A WSRP portlet is not a Web service in its own right (it cannot be accessed directly, on the contrary it must be accessed through its parent producer).
- **WSRP consumer:** This is a Web service client that invokes producer-offered WSRP Web services and provides an environment for users to interact with portlets offered by one or more such producers. The most common example of a WSRP consumer is a portal.

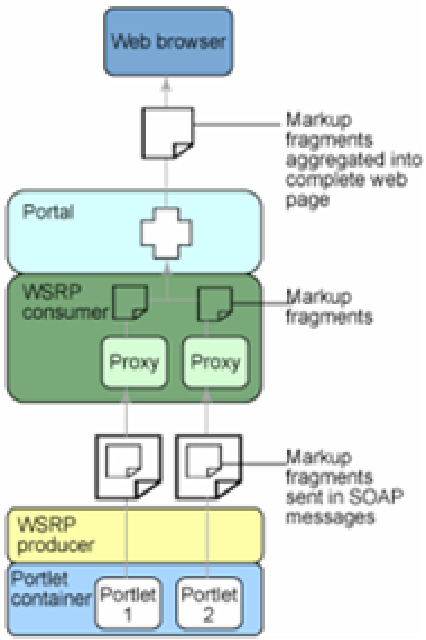


Figure 32. DILIGENT Portal Architecture

There exist different implementations for WSRP producer and WSRP consumer. In DILIGENT we use WSRP4J [31] created by Apache, which is an open source project. WSRP4J is a platform for developing and hosting WSRP compliant web services.

Having clarified these points, it is clear that each DILIGENT portlet must be WSRP compliant and therefore it can be deployed on a DHN equipped with the WSRPJ environment.

5.5.2 General-purpose services

In this section we briefly present some general-purpose services designed to equip the DILIGENT portal with the basic user functionality, i.e. login, logout, digital object management, search and browse, portal administration. As already done for all the other services, we describe them from an architectural point of view, i.e. in terms of their main components. Further details can be found in [21].

5.5.2.1 Login/Logout Elements

This set of elements yields the functionality allowing users to login/logout to the DILIGENT system. Depending on the user rights, the login allows accessing the DILIGENT/DL's functions and resources.

From an architectural point of view, it is designed as a portlet whose methods can be divided into two sets:

Control/Display Methods: These are responsible for invoking other methods, composing the results and creating the web page. All of them are inherited by GenericPortlet (processAction(), doView(), etc).

Helper Methods: These methods communicate with other services, execute computational effort or construct part of the final web page. Generally, their use is to help control/display methods to perform their actions.

In this particular portlet, helper methods' effort is to check if user is valid (password is correct), and to create a session attribute that keeps information about the user, e.g.his/her username, a Short Term Credential, personal preferences. When logout is performed, this attribute is removed from the session, and the user's files, stored in the temporary storage, are deleted.

5.5.2.2 Digital Objects Management

Digital Objects is the name used by the Application-Specific Services to refer the information objects managed by the Content Management Services (see Section 5.2) as they are visualized to the end-users.

In DILIGENT there are two different contexts where the digital objects exist in:

1. in the first context the objects are globally stored (via the Content Management Service, see Section 5.2.1) and they can be searched from the Index & Search Services (see Section 5.3) (*global objects*).
2. the second context is the portal context where users can store temporary files for future use. For instance, a user may want to upload a file in order to perform a similarity search, or to add it in an existing collection (*portal objects*).

Depending on the context, different components are provided.

For the management of global objects:

- **ManageCollectionsPortlet** (Portlet) – It provides the user interface. Also, it contains the logic (by relying on the Content Management functionality) to manage global objects, e.g. access, upload, copy, linking, and delete, in the context of an identified collection.

For the management of portal objects:

- **ManageLocalObjectsPortlet** (Portlet) – It is the user interface for upload and manipulation of the "portal objects".

These portlets are also designed with the logic of control/display methods and helper methods.

5.5.2.3 Search and Browse Service

Search and Browse are probably the most exploited functionality of any Digital Library system. This service is in charge of providing the user interface and the facilities to appropriately use the powerful search facilities offered by the Search Service described in Section 5.3.1.

From an architectural point of view it is worth noting that this service is composed by three components/portlets:

- **SubmitSearchPortlet** (Portlet) – It provides the user interface for submitting a search, i.e. create new search criteria, call search service to execute the query.
- **BrowsePortlet** (Portlet) – It provides the user interface and the back-end logic for brwsing the query's results. Its main goal is to provide navigational functionalities, i.e. retrieve next/previous page of the results.
- **DrillInPortlet** (Portlet) – It enables users to see the actual digital object they wish.

5.5.2.4 Visualization Service

This service provides a *VisualizationPortlet*, a portlet capable to adapt its rendering action to the type and characteristics of the object to be visualized. In particular, the portlet provided by this area deals with the visualization of important objects, e.g. two-dimensional charts, images and maps, and trees.

5.5.2.5 Portal Administration

As stated, the portal is configurable. Certain aspects of configuration are pre-existing to the portal deployment and performed via the editing of appropriate configuration files while others can be executed during the portal lifetime and performed via an appropriate user interface. The GridSphere is already equipped with a portlet supporting the specification of the theme (e.g. graphics, colours, logos) the portal uses, the portlets to be showed in each tab, the language the portal labels are expressed in. This basic functionality is enriched and customized on the needs of the DILIGENT project and its user communities.

6 CONCLUSION

This report has presented the DILIGENT software architecture. After clarifying the role software architecture, we have introduced and modeled three concepts of general purpose, i.e. *Resource*, *User*, and *Virtual Organization*. Then, a clear picture of the concrete integration and exploitation of grid technologies into the context of the system architecture has been provided. Finally, the highest-level concept of a DILIGENT system in terms of the services forming it and their organization into a whole and cooperating entity has been presented by highlighting the service-to-service interactions in providing the expected functionality.

This software architectural specification is given in the formal notation recommended by the Unified Process software engineering methodology, i.e. a set of UML diagrams accompanied by texts as expressed by the DILIGENT technological partners.

All partners have contributed to the production of this deliverable. The contributions have been collected, analyzed, and integrated by the CNR-ISTI team, leader of Task T1.1.2, with the support of all the other partners mainly during the validation phase of the document in order to guarantee that this document represent the understanding of the whole set of institutions forming the DILIGENT consortium.

References

[1] DILIGENT. Deliverable No D1.1.1: "Test-bed Functional Specification"

[2] D. Garlan, M. Shaw. *An Introduction to Software Architecture*. In Advances in Software Engineering and Knowledge Engineering, I (Ambriola V, Tortora G, Eds.) World Scientific Publishing Company, 1993.

[3] W3C Web Services Activity web site. http://www.w3.org/2002/ws/

[4] I. Foster, J. Geisler, W. Nickless, W. Smith e S. Tuecke. *Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment* in Proc. 5th IEEE Symposium on High Performance Distributed Computing. pp. 562-571, 1997

[5] M. Atkinson, D. DeRoure, A. Dunlop, G. Fox, P. Henderson, T. Hey, N. Paton, S. Newhouse, S. Parastatidis, A. Trefethen, and P. Watson. *Web Service Grids: An Evolutionary Approach*. http://omii.ac.uk/WSG/WebServiceGrids.pdf

[6] F. Berman, G. Fox, and A. Hey, *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, April 2003. ISBN: 0470853190.

[7] I. Foster, C. Kesselman, and S. Tuecke. *The Anatomy of the Grid: Enabling Scalable Virtual Organization*. The International Journal of High Performance Computing Applications, 15(3):200-222, 2001.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. *The Physiology of the Grid: An Open Grid Service Architecture for Distributed Systems Integration*. Open Grid Service Infrastructure Working Group, Global Grid Forum, June 2002.

[9] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, Second edition, November 2003. ISBN: 1558609334.

[10] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. F. Ferguson, F. Leymann, M. Nally, T. Storey, W. Vambenepe, and S. Weerawarana. *Modeling Stateful Resources with Web Services*. White paper, 2004.

[11] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukin, D. Snelling, S. Tuecke, and W. Vambenepe. *The WS-Resource Framework*. White paper, 2004.

[12] JSR168: Portlet Specifications. http://jcp.org/en/jsr/detail?id=168

[13] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. *Pattern-Oriented Software Architecture.* John Wiley & Sons ISBN 0-471-95869-7, 1996

[14] Ali Arsanjani. *Service-oriented modeling and architecture*. IBM developerWorks. http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/

[15] Object Management Group. *Model Driven Architecture.* http://www.omg.org/mda/

[16] Object Management Group. *Model Driven Architecture Guide version 1.0.1.* http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf

[17] DILIGENT. Deliverable No D1.2.2: "DL Creation & Management Services Specification Report"

[18] DILIGENT. Deliverable No D1.3.2: "Content & Metadata Management Services Specification Report"

[19] DILIGENT. Deliverable No D1.4.2: "Index & Search Services Specification Report"

[20] DILIGENT. Deliverable No D1.5.2: "Process Management Services Specification Report"

[21] DILIGENT. Deliverable No D1.6.2: "User-Community Specific Applications Specification Report"

[22] S. Andreozzi (INFN) et al. *GLUE Schema Specification, version 1.2, 3 Dec. 2005* http://infnforge.cnaf.infn.it/glueinfomodel/uploads/Spec/GLUEInfoModel_1_2_final.pdf

[23] EGEE. *EGEE Middleware Architecture*. EU Deliverable DJRA1.1, July 2004. https://edms.cern.ch/file/476451

[24] EGEE. *Design of the EGEE gLite middleware external interfaces*. EU Deliverable DJRA1.2, September 2004. https://edms.cern.ch/file/487871

[25] EGEE. *Software and associated documentation*. EU Deliverable DJRA1.3, April 2005. https://edms.cern.ch/document/567624

[26] The Globus Alliance. *A Globus Toolkit Primer.* Draft version February 2005. http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/key/
[27] Borja Sotomayor. *The Globus Toolkit 4 Programmer's Tutorial*. http://gdp.globus.org/gt4-tutorial/
[28] S. Farrell, R. Housley: RFC 3281: An Internet Attribute Certificate Profile for Authorization, http://www.faqs.org/rfcs/rfc3280.html.
[29] S. Graham, D. Hull, B. Murray: Web Services Base Notification 1.3, http://www.oasis-open.org/committees/download.php/13488/wsn-ws-base_notification-1.3-spec-pr-01.pdf
[30] Web Services for Remote Portlets Specification by OASIS. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp
[31] WSRP4J Official Site. http://ws.apache.org/wsrp4j/