d4SCIENCE

Project acronym	D4Science
Project full title	**DI**stributed col**L**aboratories **I**nfrastructure on **G**rid **En**abled **T**echnology **4** **Science**
Project No	212488

Deliverable No DJRA1.1a

D4Science System High-level Design

June 2008

DOCUMENT INFORMATION

Project	
Project acronym:	D4Science
Project full title:	**DI**stributed col**L**aboratories **I**nfrastructure on **G**rid **EN**abled **T**echnology **4 Science**
Project start:	1st January 2008
Project duration:	24 months
Call:	INFRA-2007-1.2.2: Deployment of e-Infrastructures for scientific communities
Grant agreement no.:	212488
Document	
Deliverable number:	DJRA1.1a
Deliverable title:	D4Science System High-level Design
Contractual Date of Delivery:	June 2008
Actual Date of Delivery:	5 September 2008
Editor(s):	L. Candela, P. Pagano
Author(s):	S. Boutsis, L. Candela, P. Di Marcello, P. Fabriani, L. Frosini, G. Kakaletris, P. Koltsida, L. Lelii, D. Milano, P. Pagano, P. Polydoras, P. Ranaldi, F. Simeoni, M. Simi, V. Tsagkalidou, A. Turli
Reviewer(s):	P. Andrade
Participant(s):	CNR, NKUA, ENG, BDM-USTRATH, UNIBASEL
Work package no.:	JRA1
Work package title:	Overall Planning and Development Coordination
Work package leader:	CNR
Work package participants:	CNR, NKUA, ENG, BDM-USTRATH, UNIBASEL
Est. Person-months:	5
Distribution:	Restricted
Nature:	Report
Version/Revision:	1.0
Draft/Final	Final
Total number of pages: (including cover)	194
Keywords:	System Reference Architecture, Service Design, Refererence Architecture, gCube, gCore

CHANGE LOG

Reason for change	Issue	Revision	Date
The first version of the document is produced.	1	0.5	July '08
Version 0.6 resulting from the internal review of the integrated version is produced. This is the candidate version to be reviewed by the appointed reviewer.	1	0.6	July '08
Version 1.0 resulting from the revision and improvement of the previous integrated version to reply to reviewer comments and suggestions.	1	1.0	August '08

CHANGE RECORD

Issue: 1 Revision: 0.5

Reason for change	Page(s)	Paragraph(s)
All the contribution have been received and integrated	All	All

Issue: 1 Revision: 0.6

Reason for change	Page(s)	Paragraph(s)
To accommodate editor comments	All	All
Changes to the Virtual Organisation Management section to make its content homogeneous with respect to the rest;	All	Section 5.3
Changes to the Broker and Matchmaker section to make its content homogeneous with respect to the rest;	All	Section 5.5
Changes to the Process Management section to make its content homogeneous with respect to the rest;	All	Section 5.6
Changes to the Information Organisation section to make its content homogeneous with respect to the rest;	All	Sections 6.1, 6.2, 6.3 and 6.4
Changes to the Information Retrieval section to make its content homogeneous with respect to the rest;	All	Sections 7.3, 7.4 and 7.5

Issue: 1 Revision: 1.0

Reason for change	Page(s)	Paragraph(s)
To accommodate reviewer comments.	All	All

DISCLAIMER

This document contains description of the D4Science 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. E-mail: info@d4science.research-infrastructures.eu

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 D4Science 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 27 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/)

D4Science is a project partially funded by the European Union

LIST OF ABBREVIATIONS

ABE	Annotation Back End
ADL	Advanced Distributed Learning
AFE	Annotation Front End
AIS	Archive Import Service
API	Application Programming Interface
ASL	Application Support Layer
BM	Broker and Matchmaker
BPEL	Business Process Execution Language
CE	Computing Element
CMS	Content Management Service
CORI	Collection Retrieval Inference Network
CS	Compound Service
D4Science	DIstributed colLaboratories Infrastructure on Grid Enabled Technology 4 Science
DIR	Distributed Information Retrieval
DPM	Disk Pool Manager
DTS	Data Transformation Service
EC	European Commission
EPR	EndPoint Reference
gCF	gCube Core Framework
GFAL	Grid File Access Library
gHN	gCube Hosting Node
GPL	General Public Licence
GUI	Graphical User Interface
GUID	Global Unique Identifier
GWT	Google Web Toolkit
IO	Input / Output
IR	Information Retrieval
IS	Information Service
JDBM	Java DataBase Manager
JSP	Java Server Pages
JSR	Java Specification Request
LGPL	Lesser General Public Licence
LMS	Learning Management System

MD5	Message-Digest algorithm 5
OASIS	Organization for the Advancement of Structured Information Standards
PES	Process Execution Service
POS	Process Optimisation Service
PKI	Public Key Infrastructure
SCORM	Shareable Content Object Reference Model
SE	Storage Element
SRM	Storage Resource Manager
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VO	Virtual Organisation
VOMS	Virtual Organisation Membership Service
VRE	Virtual Research Environment
WS	Web Service
WSRF	Web Service Resource Framework
XENA	eXecution ENgine API
XSL	XML Stylesheet Language
XSLT	XSL Transformations

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

D4Science is a complex and innovative *e-Infrastructure* which is capable to support a powerful notion of *resources sharing* and *Virtual Research Environments*. This report documents the design of the software system, named gCube, which has been used to realise such e-Infrastructure, by introducing the Reference Model and the Reference Architecture governing it. The design of the gCube Core Framework, the gCube Infrastructure Enabling services, the gCube Information Organisation services, the gCube Information Retrieval services and the gCube Presentation services are presented.

EXECUTIVE SUMMARY

D4Science is an innovative e-Infrastructure supporting a very powerful notion of *resources sharing* that enlarges both the range of shareable resources and the context in which sharing occurs with respect to the current e-Infrastructures. Morevoer, a D4Science distinguishing feature is its support for the design, creation, and management of *Virtual Research Environments*. Objective of this report is to describe the blueprint for the overall gCube software that implements the D4Science view. This report will manifest in three different and incremental versions reflecting the design of the system at different points in time. This version is the first one and it documents the constituents of this system as designed at project month 6.

This report is organised as follows.

Section 1 introduces the document by discussing its rationale.

Section 2 presents the D4Science *Reference Model*, i.e. the set of concepts and relations among them characterising this system.

Section 3 introduces the D4Science *Reference Architecture*, i.e. the overall design pattern adopted to implement the concepts and relations identified by the Reference Model. In particular, it presents the D4Science main system constituents (gCore Framework, ~~gCore Container,~~ gCube Infrastructure Enabling Services and Application Services – Information Organisation, Information Retrieval and Presentation Services) and their interdependencies to provide the reader with a clever picture of the whole system.

Section 4 presents the principles governing the gCore Framework as well as its main features. This framework is the substantial extension of Java WS Core technology tailored to satisfy the needs of the gCube services.

Section 5 presents the gCube technology supporting the operation of the whole infrastructure as well as the dynamic creation of Virtual Research Environments, i.e. the gCube Infrastructure Enabling Services. The principles governing the design of such technology as well as the main features supported are discussed.

Section 6 introduces the gCube technology supporting the management of Information Objects including the standard Repository-oriented facilities like creation, storage, access implemented by relying on (*i*) a rich notion of Information Objects and (*ii*) a very distributed operational environment including Grid storage facilities, i.e. the gCube Information Organisation Services. The design of such a set of framework is presented as well as their characterising features.

Section 7 presents the gCube technology supporting the retrieval of Information Objects by discussing the design principle of the very open and customisable framework envisaged to satisfy the needs of very dynamic, heterogeneous and evolving scenarios like the one gCube is requested to operate, i.e. the gCube Information Retrieval Services.

Section 8 introduces the gCube technology designed to provide its end-users with a highly integrated and customisable environment making gCube facilities available through a portal. Design principles, envisaged constituents and resulting features are discussed.

Section 9 concludes the document by summarising the overall main features and providing hints for future consolidation and enhancement activities of the system.

1 INTRODUCTION

D4Science provides an innovative and complex "system" that results from (*i*) an already existing software system, i.e. the gCube framework [20], and (*ii*) an activity dedicated to consolidate and enhance such existing system originally designed to operate in a test-bed scenario. Moreover, because of the integration with gLite [22], such system is capable to exploit the largest Grid infrastructure currently existing, EGEE [15].

D4Science as a system falls in the category of *Service Oriented Infrastructures*, i.e. the application of the Service Orientation principles in realising an infrastructure. This reflects in facilities for the reusability and dynamic allocation of the resources forming the infrastructure itself. In addition, D4Science supports a very flexible and agile application development model based on the notion of *Software as a Service* (*SaaS*) [7] in which components may be bound instantly, just at the time they are needed and then the binding may be discarded. Such development model materialises in the support for *Virtual Research Environments* (VREs) (cf. Section 2.5) representing a distinguishing feature of D4Science. According to it, user communities are enabled to define their own applications by simply selecting the application constituents (the services, the collections, the machines) among the pool of resources made available through the D4Science e-Infrastructure. The cost of operating each defined VRE is completely outsourced to the infrastructure that by applying *economies of scale* to the operation of the applications (sharing and re-use) can offer better, cheaper and more reliable applications than single communities can themselves.

To implement such a scenario a Service Oriented Software System has been designed and implemented during the DILIGENT EU IST project [13], the gCube system [20]. The design of such framework has been thoroughly documented in a series of deliverables containing the detailed design of all constituents[1] ([2][5][35][33][18][42]). However, despite the very powerful support for Virtual Research Environments provided by gCube, D4Science is called upon to serve a different and more demanding scenario by guaranteeing a production level infrastructure. This imposes a *consolidation* and *expansion* activity of the gCube System. Such an activity will be transversal and somehow invasive, i.e. potentially all the gCube constituents (including algorithms and mechanisms) have to be thoroughly evaluated both with respect to functional and non-functional requirements (e.g. performance, dependability, resilience) and consequently re-arranged.

The objective of this report is to document the overall D4Science "system" and in particular the gCube Framework guaranteeing its operation. The principles governing the design of the overall system as well as of the single constituents are discussed.

The report will manifest in three different and incremental versions (DJRA1.1a at project month 6 - June 2008, DJRA1.1b at month 12 - December 2008 and DJRA1.1c at month 18 - June 2009) reflecting the fact that the system will be developed incrementally since the design phase. This version is the first one and it documents the constituents of this system as designed at project month 6.

1.1 Document Structure

The remainder of this report is organised to describe the design of the whole D4Science "system" to the reader through different *views* each adding more detail to the previous. Section 2 introduces the D4Science *Reference Model*, i.e. the concepts and relations among them characterising the overall system. Section 3 describes the gCube *Reference Architecture*, i.e. the overall design model adopted to implement the constituents

[1] The gCube Framework consists of 22 subsystems and 166 components (more than 60 web services, libraries, third parties software, portlets, etc).

identified by the Reference Model. In particular, it provides the reader with a picture of the whole system in a nutshell. Having introduced the overall picture, the rest of the sections focus on providing the reader with details about the various parts of the system. Section 4 presents the gCore Framework as well as its main features. Section 5 presents the gCube Infrastructure Enabling Services, i.e. the gCube technology designed to support the operation of the infrastructure as well as the dynamic creation of Virtual Research Environments. Section 6 introduces the gCube Information Organisation Services, i.e. the gCube technology supporting Information Objects management including the standard Repository-oriented facilities like creation, storage, access implemented by relying on (i) a rich notion of Information Objects and (ii) a very distributed operational environment including Grid storage facilities. Section 7 presents the gCube Information Retrieval Services, the gCube technology supporting the *retrieval* of Information Objects. The very open and customisable framework supporting the retrieval of heterogeneous Information Objects from different and distributed data sources through a broad set of methodologies and approaches ranging from Google-like queries to geo-spatial queries is discussed in detail. Section 8 introduces the gCube Presentation Services, i.e. the gCube technology designed to provide end-users, both human and inanimate, with an highly integrated and customisable environment making all gCube facilities available through a single access point. Finally, Section 9 summarises the overall main features and provides hints for future consolidation and enhancement activities of the system.

The gCube system follows a Service Oriented Approach, thus its main constituents are implemented by ***Services*** (actually Web Services either stateful or plain). Each constituent Service is documented in terms of (*a*) the ***resource*** it manages, (*b*) the ***properties*** about the managed resources it exposes, and (*c*) the ***functions*** it supports, i.e. the main actions that can be accomplished through the usage of the service. It is worth noting that such operations are envisaged at a high-level design and can correspond to one or more methods in the concrete design and implementation, e.g. the WSDL describing the Web Service API.

In addition to Services, the D4Science system is equipped with ***Software Libraries*** and ***Portlets***. The former are documented in terms of the supported ***functions***, the latter are documented in terms of the design of the ***GUI*** they implement and the ***functions*** they support.

2 THE D4SCIENCE SYSTEM REFERENCE MODEL

According to [31] 'A reference model is an *abstract framework* for understanding significant relationships among the entities of some environment. It enables the development of specific *reference* or *concrete architectures* using consistent standards or specifications supporting that environment. A reference model consists of a minimal set of unifying concepts, axioms and relationships within a particular problem domain, and is independent of specific standards, technologies, implementations, or other concrete details'. This section introduces the concepts characterising D4Science as a system. Before to proceed, it is fundamental to clarify the fact that the D4Science "system" is actually composed of two different materialisation of the term system, the "*living system*" users interact with (a.k.a. the D4Science Infrastructure) and the "*software system*" supporting the deployment and operation of the D4Science Infrastructure (a.k.a. gCube [20]). Clearly, these systems are closely related, the software system is the enabling technology of the living system. This report will focus on the "software system" but the concepts presented in this reference model characterise both "systems".

The concepts constituting the D4Science Reference Model are organised in different domains: the ***resource domain*** represents the entities managed by gCube system; the ***user domain*** represents the entities, both human and inanimate, interacting with the system; the ***policy domain*** represents the rules governing the functioning of the system; the ***architecture domain*** represents the concepts and relations needed to characterise a gCube based architecture; the ***VRE domain*** represents the concepts and relations needed to characterise a *Virtual Research Environment*.

2.1 Resource Domain

The gCube system is a software system conceived to manage an infrastructure consisting of a set of heterogeneous entities.

All such heterogeneous ***resources*** share some commonalities (*gCubeResource)*:

- Each gCube resource has a ***unique identifier*** (ID);
- Each gCube resource has a ***type*** (Type) allowing to discriminate and capture the role/semantic such resource is supposed to play;
- Each gCube resource has ***multiple scopes*** (Scopes) allowing to characterise the contexts the resource is supposed to operate (VO/VRE);
- Each gCube resource has a ***profile*** (Profile) to capture the distinguishing features of the resource to support resource discovery and usage.

In Figure 1 the class diagram capturing this domain is presented.

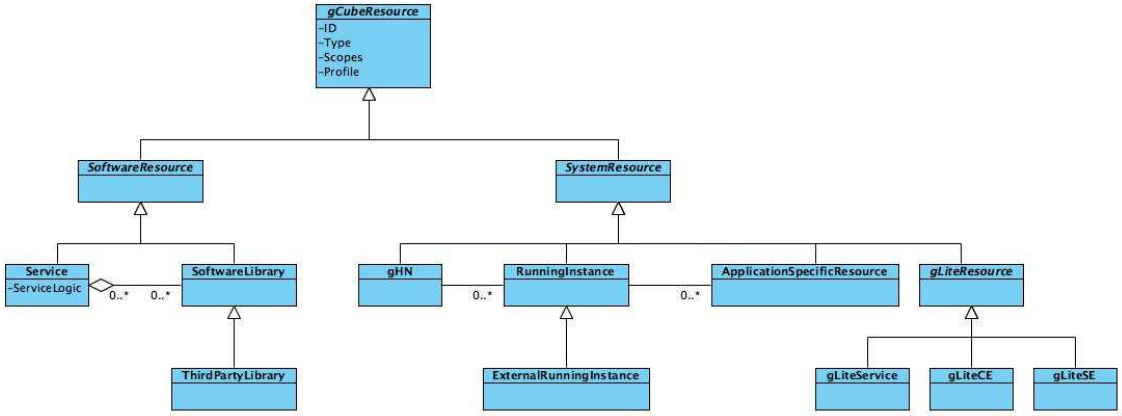

Figure 1. D4Science Resource Model

Two abstract classes characterise this domain, the *SoftwareResource* and the *SystemResource*. The former is to capture the resources forming the software managed by the gCube system, the latter is to capture the rest of resources managed by the gCube system, e.g. the hosting nodes, the running services. This distinction is justified by one of the distinguishing feature of gCube, i.e. the capability to dynamically deploy software components to produce new resources by relying on other existing resources (cf. Section 2.4). Thus, the software forming gCube becomes itself a resource managed through gCube.

For what concerns *SoftwareResources* the following resource typologies exist:

- ***Service***, a *SoftwareResource* delivering its expected functionality through a web based interface. In a Service Oriented Architecture (SOA) it is a constituent unit of the system. gCube exploits the SOA paradigm and implement it by relying of the WSRF framework [6]. Each service is comprised of a software (*ServiceLogic*) implementing the service-specific business logic and zero or more *SoftwareLibrary* acting as helper software implementing non-service-specific logic, i.e. piece of software implementing general purpose functions, e.g. XML parse functions.
- ***SoftwareLibrary***, a *SoftwareResource* delivering its expected functionality in a stand-alone manner via a programmatic API. It is important to model such piece of code as resource in order to promote the reuse.
- ***ThirdPartyLibrary***, a *SoftwareLibrary* delivering its expected functionality in a stand-alone manner via a programmatic API. This specialisation of *SoftwareLibrary* is due to the need to capture the peculiarity of such software at deployment time, i.e. the fact that such a piece of software has its own deployment procedures.

For what concerns *SystemResources* the following resource typologies exist:

- ***gLiteResource***, a *SystemResource* representing a gLite resource [22], i.e. a placeholder in the gCube infrastructure for a resource forming a gLite based infrastructure. It is further specialised in ***gLiteService***, ***gLiteCE*** and ***gLiteSE*** to capture the main types a gLiteResource can be.
- ***gHN***, a *SystemResource* representing the hosting machine on which gCube can dynamically deploy a *Service* (along with all the needed *SoftwareLibraries*) to create a *RunningInstance*.
- ***RunningInstance***, a *SystemResource* representing a *Service* deployed in a *gHN*. It is the runtime manifestation of a *Service* and consequently the runtime implementation of the expected *Service* facilities.
- ***ExternalRunningInstance***, a *RunningInstance* representing an instance of a *Service* running outside the direct control and management of gCube, i.e. (re-)deployment of such a *Service* is not allowed since gCube does not manage the *Service*. An example of such a kind of *RunningInstance* is an up and running Web Service, e.g. one of the services forming the G-POD application[2], whose facilities are needed in a VRE;
- ***ApplicationSpecificResource***, a *SystemResource* representing a resource created and managed by a specific Service, e.g. a Collection managed by the CollectionService (cf. Section 6.4.3), a TransformationProgram managed by a MetadataBroker Service (cf. Section 6.7.1.1.1).

2.2 User Domain

In gCube the term "user" identifies entities entitled to interact with the infrastructure. This definition not only includes human users (externals to the infrastructure) but also services (part of the infrastructure) autonomously acting in the system. Batch services (e.g. monitoring services) are an example of such users that, reacting to status changes or on a temporal basis, interact with other entities. Thus, some gCube services need to be identified and managed, from the authentication and authorization point of view, as users.

[2] http://eogrid.esrin.esa.int

gCube architecture exploits the Public Key Infrastructure (PKI) paradigm to uniquely identify users in the infrastructure. gCube users are provided with a private key and a public certificate that can be used to authenticate to other entities. Private keys and public certificates are issued and revoked to users by a trusted entity, named Certification Authority (CA). The gCube infrastructure is designed to support CAs acknowledged by the International Grid Trust Federation (IGTF) [27] as well as an infrastructure-specific CA, named D4Science CA.

VO-Management components (detailed in section 5.3) provide functionalities to manage gCube users and their credentials.

2.3 Policy Domain

In gCube, the **Virtual Organization** (VO) concept is used to define authorization policies in the infrastructure. A Virtual Organization is a dynamic pool of distributed resources shared by dynamic sets of users from one or more real organizations. Resource Providers (RP) usually make resources available to other parties under certain **sharing rules**. Users are allowed to use resources under Resource Provider (RP) conditions and with the respect of a set of VO policies.

Following this approach, in the gCube VO model a policy is defined as a **permission** for a **user** to perform an **operation** on a specific **resource**. In the model resources are univocally identified through a resource id and must belong to a **resource type**. Each resource type is associated to a set of logical operations. These operations can be performed over resources of that type in that model. It is worth notice that operations in the VO model are just identifiers used to define logical operations that can be performed over resources (e.g. read, modify, delete). They not necessarily identify methods exposed by resource implementation (e.g. get, put). Logical operations are useful to describe logical operations a resource exposes and map to methods provided by resource implementation at the resource side.

The gCube VO model also leverages the concept of **role**, as introduced by the RBAC model [17], to decouple the association between users and permissions. Furthermore, roles are organized in hierarchies, thus allowing a natural way to capture organizational lines of authority and responsibility. Role hierarchies are not constrained to be trees; each role can have several ancestors with the only constraint that cycles are not allowed in the structure.

The gCube VO model is shown in Figure 2.

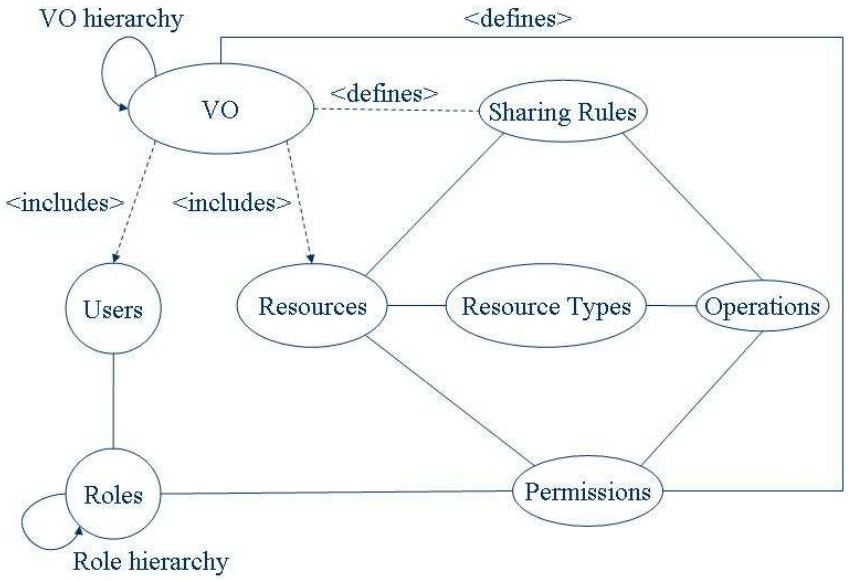

Figure 2. The gCube VO Model

The model takes into account two different actors managing policies over resources:

- **Resource Providers** – users defining sharing rules, i.e. policies to resource access w.r.t. Virtual Organizations as a whole;
- **VO Managers** – users defining permissions, i.e. policies to access resources w.r.t. VO members individually. Permissions have to comply with sharing rules defined by Resource Providers for a specific VO.

VO-Management components implement the gCube VO model as detailed in section 5.3.

2.4 Architecture Domain

The gCube system relies on a component-oriented Architectural model which subsumes a '***component-based approach***', i.e. a kind of application development in which:

- The system is assembled from discrete executable components, which are developed and deployed somewhat independently of one another, and potentially by different players.
- The system may be upgraded with smaller increments, i.e. by upgrading some of the constituent components only. In particular, this aspect is one of the key points for achieving interoperability, as upgrading the appropriate constituents of a system enables it to interact with other systems.
- Components may be shared by systems; this creates opportunities for reuse, which contributes significantly to lowering the development and maintenance costs and the time to market.
- Though not strictly related to their being component-based, component-based systems tend to be distributed.

The components characterising a gCube based system from an architectural point of view have been introduced in the Resource Domain section (cf. Section 2.1): all the *gCubeResource* types with the exception of the *ApplicationSpecificResource* are considered first class citizens in a component-oriented architecture. However, the Architecture Domain itself can be described through different and focused views each capturing a different facet of this multifaceted domain. Each of these views potentially uses a subset of the architectural components. For instance, if the goal of the view is to describe the current running instance of the gCube based Infrastructure serving D4Science the concepts that will be primarily used are the notions of: *gHN* (to capture the machines implementing the Infrastructure), *RunningInstance* and *ExternalRunningInstance* (to capture the running services supporting the operation of the whole), and *gLiteResource* (to capture the nodes of a gLite based infrastructure conceptually contributing to the whole) are used. On the other hand, if the goal of the view is to describe the system from a static point of view, i.e. its software constituents, the notions of *Service*, *SoftwareLibrary* and *ThirdPartyLibrary* are used.

2.5 VRE Domain

The notion of Virtual Research Environment is a cornerstone of the whole D4Science Infrastructure.

From a user perspective, a Virtual Research Environment is an integrated and coordinated working environment providing participants with the resources (data, instruments, processing power, communication tools, etc.) they need to accomplish the envisaged task. This environment is expected to be particularly dynamic because of the potential dynamicity of the participating community, both in terms of the constituents as well as of their requirements. In fact, it is envisioned to serve the e-Science vision that demands for the realisation of research environments enabling scientists to collaborate on common research challenges through a seamless access to all the resources they need regardless of their physical location. The resources shared can be of very different nature and vary across application and institutional domains. Usually they include *content resources*, *application services* that manipulate these content resources to produce new knowledge, and *computational resources,* which physically store the content and support the processing of the services.

From a system point of view, a VRE is a pool of gCubeResources dynamically aggregated to behave as a unit w.r.t. the application context the VRE is expected to serve. Each VRE is a ***view*** over the potentially unlimited pool of resources made available through the Infrastructure that (*i*) is regulated by the user community needs and the resources sharing policies and (*ii*) produces a new VO constraining the scope and usage of resources actors playing in the VRE are subject to.

3 THE GCUBE SYSTEM REFERENCE ARCHITECTURE

A *Reference Architecture* is an architectural design pattern that indicates how an abstract set of mechanisms and relationships realises a predetermined set of requirements [31].

The gCube system captured by the Reference Architecture in Figure 3 is the Software System resulting by combining in a Service Oriented Architecture a number of subsystems[3] .

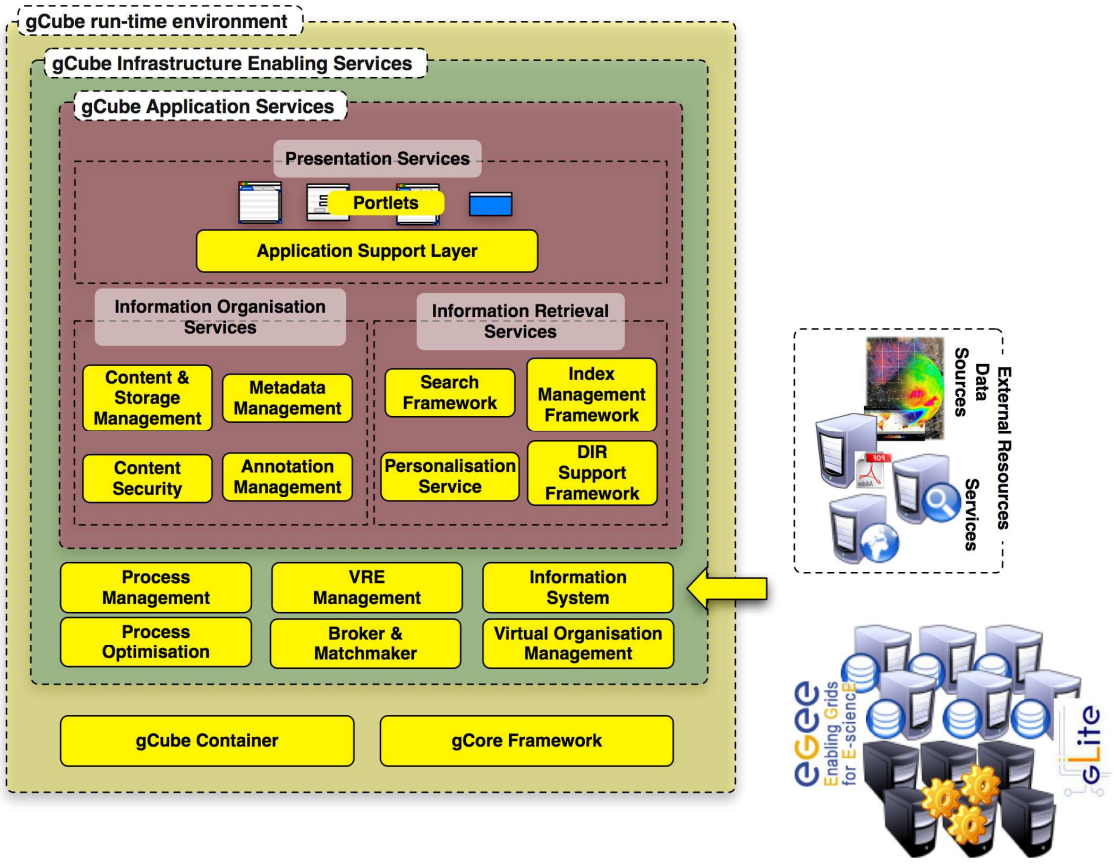

Figure 3. D4Science System Reference Architecture

Such subsystems are organised in a three-tier architecture consisting of:

- the ***gCube run-time environment,*** named gCube Hosting Node environment or simply gHN, is the set of subsystems equipping each gCube empowered machine and forming the platform for the hosting and operation of the rest of system constituents. Namely, it consists of the *gCube Container*[4] (to run gCube Services), the *gCore Framework, named gCF* (to reinforce the gCube Container in supporting the operation of gCube Services), a number of local services, namely Deployer (cf. Section 5.4.4), gHNManager (cf. Section 5.4.6), and Delegation (cf. Section 5.3.3), and a number of

[3] A subsystem is intended as a logical constituent unit when considered with respect to the system as a whole. A subsystem groups Services, Libraries and any other kind of software component belonging to the area of competence of the subsystem.
[4] A customisation of the WS Core Container distributed by the Globus® Alliance [36].

libraries and stubs needed to manage the communication with all other gCube services.

- the ***gCube Infrastructure Enabling Services*** is the set of subsystems constituting the backbone of the gCube system and responsible to implement (*i*) the operation of an *e-Infrastructure* supporting *resources sharing* and (*ii*) the definition and operation of *Virtual Research Environments*;
- the ***gCube Application Services*** is the set of subsystems implementing facilities for (*i*) *storage*, *organisation*, *description* and *annotation* of information in a VRE (*Information Organisation Services*), (*ii*) *retrieval* of information in the context of a VRE (*Information Retrieval Services*) and (*iii*) provision of VO and VRE users with an interface for *accessing* such an e-Infrastructure.

The overall architecture has been designed following the Service Oriented Architecture principles:

- the main constituents of each subsystem are expected to be *loosely-coupled* Web Services (actually WSRF services [6]);
- the constituents of the gCube-based e-Infrastructure will be discovered thanks to the *Information System* subsystem that, as usual, becomes fundamental to guarantee the operation of the rest;
- such loosely-coupled Services can organised in workflows as to form compound services whose orchestration is guaranteed by the *Process Management* subsystem.

It is woth noting in this reference architecture that the runtime environment is an integral part of the overall system because the management of the environment hosting the services and the management of the service lifetime is part of the gCube business logic. Thanks to the gHN capabilities, other gCube services can be dynamically deployed on remotely gHNs to serve the needs of Virtual Research Environments (further details about the dynamic deployment are provided in Section 5.4). Figure 4 presents the ***gCube Hosting Node (gHN)*** Reference Architecture.

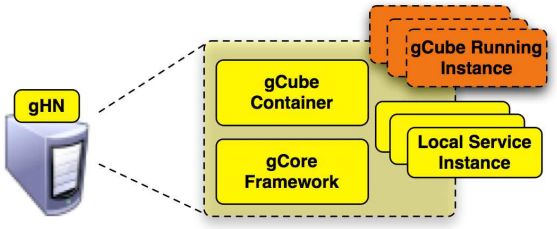

Figure 4. gHN Reference Architecture

In the remainder of this section the constituents of the Reference Architecture are introduced starting from the lower layer.

3.1 The gCore Framework

The *gCore Framework* (gCF) is a Java framework for the development of high-quality gCube services and service clients. It provides an application framework that allows gCube services to abstract over functionality lower in the web services stack (WSRF, WS Notification, WS Addressing, etc.) and to build on top of advanced features for the management of state, scope, events, security, configuration, fault, service lifetime, and publication and discovery. A detailed description of the high-level design of such a framework is presented in Section 4.

3.2 The gCube Infrastructure Enabling Services

The *gCube Infrastructure Enabling Services* is the family of subsystems implementing the foundational services that guarante the operation of the e-Infrastructure. Such functions are organised in four main areas: (*i*) organisation and execution of Virtual Research Environments (*VRE Management*) by guaranteeing an optimal consumption of

the available resources (*Broker and Matchmaker*); (*ii*) registration of the infrastructure constituents (*Information Service*); (*iii*) the authentication and authorization policy enforcement enabling the highly controlled sharing of infrastructure constituents (*Virtual Organisation Management*); and (*iv*) definition and orchestration of complex workflows (*Process Management*) by guaranteeing an optimal consumption of the available resources (*Process Optimisation*). In particular:

- the **VRE Management** services are responsible for: (*i*) the definition of VREs and (*ii*) the dynamic deployment of VRE resources across the infrastructure. VREs definitions are declaratively specified through an appropriate and user-friendly user interface in a dedicated language and inform the derivation of an optimal deployment plan. The plan is based on availability, QoS requirements, resource inter-dependencies, and VRE sharing policies, but also on monitoring of failures (resources are dynamically redeployed) and load (resources are dynamically replicated). Three distinguished services (*Software Repository*, *Deployer*, *gCube Hosting Node Manager*) support VRE definition and dynamic deployment by, respectively, collecting service implementations, deploying service implementations and their dependencies on gHN, and hosting such service implementations at selected nodes.
- The **Broker and Matchmaker** service identifies the set of gHNs where to deploy a set of services. In particular, given a set of packages to be deployed, their requirements versus the environment and/or other services, it identifies the set of gHN to be used as target hosts for the deployment action.
- the **Information Service** allows the publication of descriptive information about VRE resources, the discovery of VRE resources based on descriptive information, and the real-time monitoring of VRE resources based on subscription/notification mechanisms. Heavily relied upon all the functional layers of gCube, the Information Service is a replicated service in which instances communicate in peer-to-peer fashion to maximize availability, response time, and fault tolerance.
- the **Virtual Organisation Management** services are responsible for equipping gCube with a robust and flexible security framework for managing Virtual Organizations (VOs). gCube exploits the VO mechanism to enforce a trusted and controlled environment in each dynamically created VRE. The main features consist in user and group management, authentication support, authorization definition, delegation, and enforcement of the security credential. These services rely on and integrate VOMS [1] and Globus Security Infrastructure (GSI) [45] to provide gCube with a security framework supporting various configurations. The actors of this framework are humans as well as services.
- the **Process Management** and **Process Optimization** services support the definition and execution of processes, i.e., workflows combining gCube services, external services and gLite jobs to deliver new functionalities (also known as *programming in the large* [46]). In particular, these services provide the basic functionality for *(i)* creating processes either via a graphical *process modelling* tool or via a BPEL definition [10], *(ii)* reliably *executing* processes in a fully distributed and decentralized, thus highly scalable, way and *(iii) optimizing* processes both at build-time and at run-time. Process execution facility has been designed and implemented to take full advantages of the Grid, i.e. process steps are outsourced to the resources forming the e-Infrastructure and the process is executed in a distributed peer-to-peer modality. In particular, the Process Management service integrates the gLite [22] software, thus enabling gCube to run such processes on EGEE resources. A monitoring front-end allows to get information on individual process instances which are not materialized on a single host because of their distributed execution. This monitoring front-end allows administrators to follow the state of execution of a process instance online and also shows where the different parts are being executed.

The high-level design of the constituents of such areas is presented in Section 5.

3.3 The gCube Application Services

The gCube Application Services is the family of subsystems delivering three outstanding functions of any Virtual Research Environment: (*i*) *storage*, *description* and *annotation* of information in a VRE (*gCube Information Organisation Services*), (*ii*) *retrieval* of information in the context of a VRE (*gCube Information Retrieval Services*) and (*iii*) provision of VRE users with an interface for *accessing* such an information and the rest of functions equipping a VRE (*gCube Presentation Services*).

3.3.1 The gCube Information Organisation Services

The *gCube Information Organisation Services* is the family of subsystems implementing the foundational services guaranteeing the management (storage, organisation, description and annotation) of information by implementing the notion of *Information Objects* (cf. Section 6.1), i.e. logical unit of information potentially consisting of and linked to other Information Objects as to form *compound objects*. Such functions are organised in three main areas: (*i*) the storage and organisation of Information Objects and their constituents (*Content and Storage Management*); (*ii*) the management of the metadata objects equipping each Information Object (*Metadata Management*); and (*iii*) the management of the annotations objects potentially enriching each Information Objects (*Annotation Management*). In particular:

- the **Content & Storage Management** services provide transparent access to Information Objects managed through gCube. In particular, they provide basic functionality for: *(i)* manipulating Information Objects and/or collections, i.e. creating, accessing, storing, and removing; *(ii)* orchestrating distributed storage nodes and providing a transparent access to them; *(iii)* a notification mechanism to maintain derived data upon changes in content; and *(iv)* importing Information Objects from different content providers through *wrappers*. The kind of Information Object manageable by the Content & Storage Management services is generic and flexible enough to model and thus support several content types. To make full exploitation of Grid storage facilities, the *Storage Management* service provides an abstract interface to the underlying distributed and heterogeneous actual storage interfaces and technologies (e.g. DPM [14] via SRM [34] and the GFAL interface [21]).Thanks to the gCube replication management subsystem and by integrating gLite, the gCube *Storage Management* service is capable to exploit the storage capacity of the EGEE infrastructure and maintain multiple copies of the Information Objects as to maximise IOs availability.
- the **Metadata Management** services provide functionality for managing metadata objects, i.e. additional data attached to Information Objects. In particular, these services support *(i)* the manipulation of metadata objects and metadata collections, i.e. creating, accessing, storing, and removing metadata objects compliant to one or more metadata format, *(ii)* the definition of metadata formats, *(iii)* the transformation of metadata objects into diverse formats via user-defined transformation programs, and *(iv)* the search for metadata objects. These characteristics make the services capable to manage multiple formats of metadata. Moreover, the support for diverse metadata formats and the relative transformation programs are an important feature for dealing with heterogeneity issues. To store metadata objects the services exploit the storage facilities provided by the Content & Storage Management and thus guarantee improvement in reliability and access of managed objects.
- the **Annotation Management** services are responsible for cross-model, and cross-media back-end management of annotations, a manually authored and subjective specialisation of metadata objects. The services mediate between interactive annotation front-ends and Metadata Management services by: *(i)* enforcing a consistent modelling of annotation relationships between Information Objects, and *(ii)* increasing the simplicity, granularity, and flexibility with which annotations are created, collected, deleted, updated, and interrelated as specific forms of metadata objects.

The high-level design of the constituents of such areas is presented in Section 6.

3.3.2 The gCube Information Retrieval Services

The gCube Information Retrieval Services is the family of components offering Information Retrieval (IR) facilities to the gCube infrastructure, i.e. allowing searching over data and information by a wide range of techniques. The IR family of services can be decomposed in three major categories, which are presented below and are entitled as "frameworks" due to the fact that they are not standalone services. Instead, they are rather large collaborating systems based on protocols, specifications and software, which expose remarkable extensibility to the gCube system they empower:

- **Search Framework**: This category includes all services focused on the search-specific aspects of the gCube platform. More analytically, it consists of the *search orchestrator component*, *search operators*, *query processor components* and the *data transfer mechanism*. The workflow required for computing a user-query is the following: The search orchestrator receives queries from the gCube portal, communicates with the gCore IS service for retrieving environment information. In the next step, the orchestrator feeds this information along with the query to the query processor components which ultimately produce an execution plan. This plan is forwarded to the gCube execution engine (one of which is the Process Management Service) which orchestrates the execution by invoking the search operators, as dictated by the plan. The data transfer is performed by the *ResultSet* component of the Search Framework. However, due to its importance, it requires special credit and therefore is analyzed in a distinct section (cf. Section 7.2). The final results are then forwarded to the user (portal). The Search Operators cover most of the traditional relational algebra operations, as well as some advanced ones, such as geospatial search and similarity search, thus providing a full fledged set of capabilities to the final user. Index Management and DIR frameworks provide a major part of the Search Operators and are analyzed in distinct sections. For further details regarding the Search Framework, please refer to Section 7.3.
- **Index Management Framework**: This category includes all services that are involved in the creation and management of gCube indices. Management refers to all aspects of an index lifecycle as well as support for search capabilities. In gCube a rich set of indices, such as full text, forward, feature, geospatial indices, is employed, offering a full-fledged set of storage and search capabilities regarding various data types and models. The services of Index Management Framework communicate with the Content And Storage Management services in order to acquire the data set to be indexed and also to preserve their state. They also employ the gCore IS capabilities so as to publish themselves and therefore be used by clients. For further details regarding this framework, please refer to Section 7.4.
- **Distributed Information Retrieval Support Framework**: This category includes all services which enhance and support the IR system. This framework provides higher-level IR capabilities which include content ranking, source selection and result set fusion (ranked merging of various data sets). Components of this framework communicate with the Index Management Services for statistic extraction and the IS service for information publication. Search Framework employs the advanced capabilities offered by DIR framework in order to enhance its search capabilities, by refining queries, enhancing produced search results and finally exhibiting a higher level of services. For further details regarding this framework, please refer to section Section 7.6.
- An additional component which does not belong to any of the frameworks mentioned above, but acts independently and improves the search quality, is the **Personalisation Service**. It is indirectly invoked by the Search Framework, through an appropriate wrapper, and used for enhancing user queries, by injecting additional "personalized" information. It is analyzed in Section 7.5.

In Section 7, the three frameworks mentioned above are analysed and the associations among them are presented.

3.3.3 The gCube Presentation Services

The gCube Presentation services form the logical top layer of a gCube-powered infrastructure. Their objective is twofold:

- To provide the means to build user interfaces for interacting with and exploiting the gCube system and infrastructure.
- To provide a full range of user interfaces for achieving interaction with the system, out-of-the-box.

The gCube presentation layer is based on the Application Support Layer (ASL), which is a framework that abstracts the complexity of the underlying infrastructure so that the front-end developer focuses on the objectives of presentation rather the details of the protocols and rules for interacting with the underlying (WSRF) services. The ASL exposes to the developer well known tools as session and credential management and is accessible through various interfaces (currently HTTP and JAVA-native).

On top of the ASL the developer can develop the user interface components needed for a particular application, depending on the execution environment that will host them (e.g. php web server, desktop application, application server etc).

The execution environment is normally provided by existing systems and can be powered by bare Operating Systems / Virtual Machines (e.g. desktop applications), plain html pages, dynamic web-sites (php, asp, jsp etc), portals, application servers etc.

gCube presentation layer, offers an initial set of components currently running under the JRS168 [28] specification, hosted by GridSphere portlet container [26], while, apart from gCube core services, it is based on Java and servlets technologies for offering it services. These components, which are called "portlets", and cover end-user and administrative functionalities, are briefly presented in Section 8 along with other details of the Presentation Layer and the ASL.

4 THE GCORE FRAMEWORK HIGH-LEVEL DESIGN

The *gCore Framework* (gCF) is a Java framework for the development of high-quality gCube services and service clients. Its main design goals are, in order of priority:

- to simplify and standardize all systemic aspects of service development, particularly those which relate to the fulfillment of gCube-specific requirements;
- to promote the adoption of best practices in distributed programming for concerns such as safety of concurrent access and autonomicity of behavior.

Overall, the framework sets out to offer a wide gamut of transparencies to the gCube developer, most noticeably in the areas overviewed in the rest of this Section. It should be noted that additional transparencies related to the use of auxiliary technologies for building, configuring, and deploying gCube services are offered in the context of the *gCube Core Distribution* (gCore). These, however, are not discussed here as they are not part of the framework and fall in fact outside the scope of an architectural specification.

4.1 Overall Architecture

The overall architecture of the gCore Framework is depicted in Figure 5. Being a software framework, it mainly consists of a series of packages clustering classes implementing the facilities described in the rest of the Section.

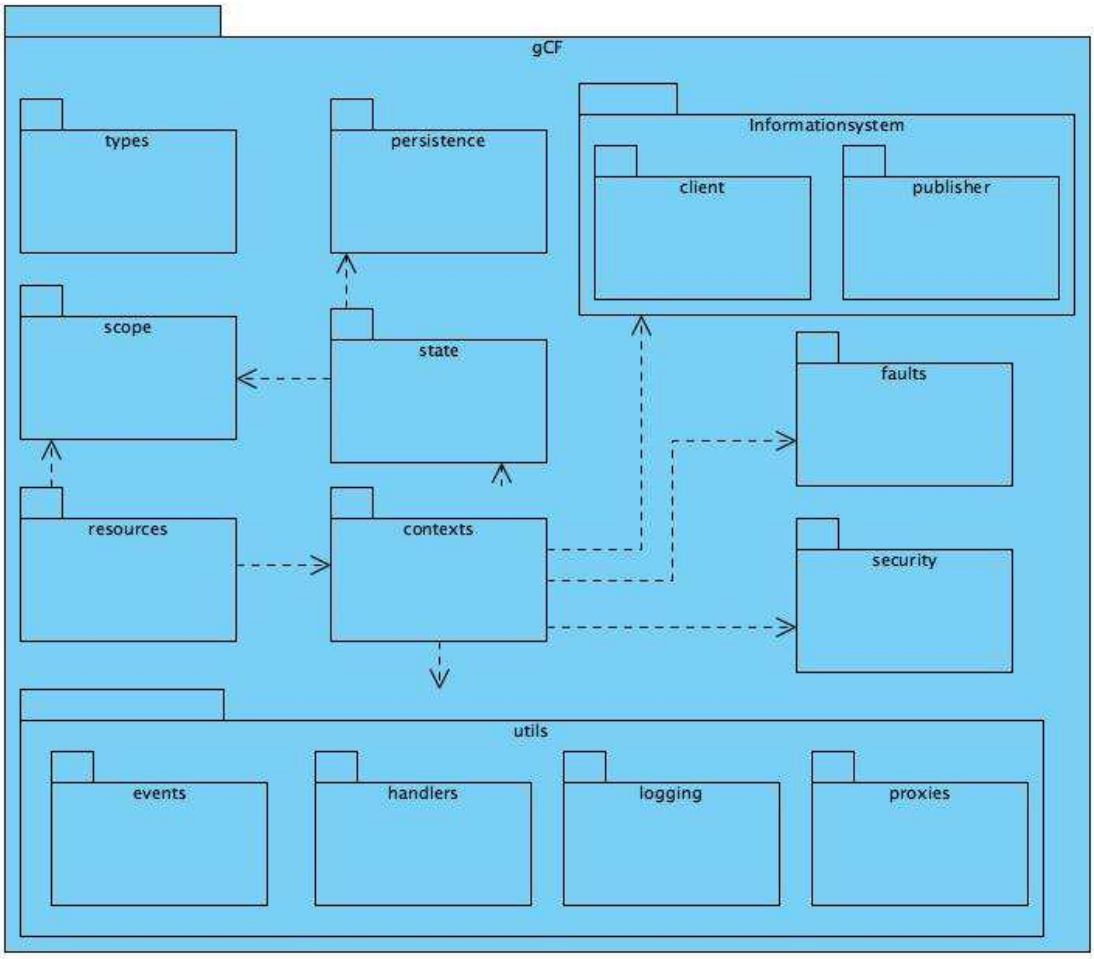

Figure 5. The gCore Framework Architecture

In particular, context is the package that implements, with the support of the other packages, the lifecycle management (cf. Section 4.2) and the configuration management (cf. Section 4.5), scope supports the scope management (cf. Section 4.3), security realises the security management (cf. Section 4.4), state supports the state management (cf. Section 4.6), faults realises the fault management (cf. Section 4.7), informationsystem implements the discovery and publication (cf. Section 4.8), event supports the event management (cf. Section 4.9), and handlers realises the local processes and remote interactions (cf. Section 4.10).

4.2 Lifecycle Management

Service instances ought to behave like finite state machines, transitioning on key events from (re-)deployment to initialization, activation, and failure. In some states they ought to engage in key activities, e.g. initialization after deployment. In others, they ought to refuse any external engagement and wait patiently for the occurrence of external events, e.g. the delegation of credentials after initialization. Some of these events may be associated with the successful completion of activities, e.g. activation in an insecure infrastructure after initialization. Others depend on external stimuli, such as the bootstrapping of the gHN or indeed its failure. Yet others are associated with the failure of other transitions. Finally, some transitions ought to be monitored and thus published within the infrastructure, e.g. activation or failure. Others are instead to be kept private, e.g. initialization.

The framework manages the entire lifetime of gCube services transparently, leaving gCube developers with the sole responsibility of communicating failure whenever they observe it. At the same time, it allows developers to customize service behaviour on each transition in accordance with service-specific semantics.

4.3 Scope Management

Hosting nodes, service instances, stateful resources, queries and more generally service calls are all associated with a notion of scope that determines their visibility and/or usability within the infrastructure. It is so that the scope of a service instance is subordinate to that the gHN in which it is deployed, or that the scope of a stateful resource is subordinate to that of the service instance which generated it. More subtly, stateful resources which are reused across different scopes ought to be published in all of them while they ought to be unpublished, but not altogether removed, when they are dismissed in some but not all of their scopes. Service calls ought to be rejected if they prove incompatible with the scope of the target service instance, while scope information ought to propagate from instance to instance in a distributed workflow and thus from thread to thread within the runtime of a single service instance.

The framework handles transparently scope bindings and scope rules enforcement for gHNs, instances, stateful resources, and calls. It also offers mechanisms to simplify propagation of call scope, from thread-safe scope managers for asynchronous scope propagation across threads which unfold within the same runtime, to stub proxies which build on scope managers to transparently propagate scope across different runtimes.

4.4 Security Management

Service instances that run within secure infrastructures may need to ascertain the identity and privileges of their clients before accepting to process (some of) their requests. If processing such requests involves interacting with other service instances - i.e. if the instances need to assume the role of callers in turn - then the instances may need to act on behalf of their clients. They may then have to make use of their identity and privileges and still comply with the security levels required for outgoing communication. On the other hand, they may need to act autonomically, and thus rely on interactions with Security services (cf. Section 5.3) for the periodic renewal and localization of credentials of their own identity as a precondition to achieve a fully operational state. Besides issues of authentication and authorization, service instance

may also lay, or else be asked to lay, certain assumptions of privacy and integrity upon communication. In the face of all these requirements, the implementation of the service ought to be largely independent from the security settings of the environment in which some of its instances might happen to operate.

The framework handles transparently security requirements of services that run in secure infrastructure, whether they need to use their own credentials and/or those of their callers. Developer specify what type of credentials ought to be used to process a given request and service-wide security managers make them asynchronously available at the points in which they are needed, before issuing outgoing calls. As for scoping, stub proxies interact autonomously with security managers to retrieve previously set credentials and set privacy and integrity settings on the outgoing calls in accordance with default requirements of the target instances. Finally, all security-related actions undertaken by framework and developers are silently disabled if the service instance happens to operate in an unsecured infrastructure.

4.5 Configuration Management

As most open technologies, those associated with Web Service development rely heavily on access to configuration resources. In Java, these include both classpath and file system resources, and may vary in their format from property files, to XML documents, to schemas for those documents. Most importantly, they tend to cover virtually all aspects of service development: from building and deployment, to logical and physical specification, to service initialization and state publication, to logging and security policies. Under assumption of dynamic service deployment, however, programmatic access to file system resources may not rely on absolute knowledge of their location. Rather, it ought to occur relatively to key locations that are dynamically resolved with respect to the gHNs on which service instances are actually deployed. Further, access to file system resource ought to be resilient to I/O failures and other forms of corruption. Finally, access to configuration resource ought to be available not only for introspection of service instances, but also for exploration and discovery of the runtime environment in which they operate. Of particular interest here may be the configuration of the gHN in which the instances are deployed, and in some advanced scenarios also the (public) configuration of co-deployed service instances.

Based on these requirements, the framework offers ground facilities to access in read or write mode arbitrary configuration resources on both classpath and file system. Write operations on the file system are implicitly associated with backup operations and, in the case of failure, read operations are transparently redirected to the available backups. Built on top of these facilities, standard configuration resources which govern building, deployment, security, publication, and similar cross-service processes are pre-parsed and exposed through dedicate channels and object bindings for convenience of inspection. Similar channels and bindings are available for access to gHN configuration. In addition, high-level access to local directory services (JNDI) extends the convenience of object bindings to idiosyncratic configuration elements, at little or no extra cost. Finally, access to the configuration of different run-time entities - e.g. gHN configuration vs. service-wide configuration vs. port-type specific configuration - is rationalized and distributed along a hierarchy of corresponding context objects.

4.6 State Management

Many service instances engage in ‘stateful’ interactions with their clients, in that their responses are not solely a function of requests but rely on some notion of application-level state that pre-exists, persists, and may evolve as a result of those interactions. State that needs to be consumed by different clients, at different times, and for different purposes requires system-wide mechanisms for its publication and discovery. Based on these, systemic solutions for state management (e.g. scheduled destruction) and state change (subscriptions and notifications) can then be implemented. Standard publication and discovery mechanisms, on the other hand, can only be built on standards for modelling state and access to state. It is thus a systemic requirement in gCube that the

design of stateful services follow the Web Service Resource Framework (WSRF [6]) set of standards for state management. In particular, stateful services ought to model state as one or more WS-Resources, i.e. first-class Web Resources which:

- aggregate local implementations of state - the *stateful resources* - with some port-type through which they can be publicly accessed;
- do so transparently, in the sense that port-types and stateful resources can be unambiguously identified from the WS-Resources which aggregate them. Precisely, WS-Resources are identified by a combination of the endpoint of their port-type and the identifier of their stateful resource (based on conventional use of *WS-Addressing* standard [9]). WS-Resource identifiers can then be used to access their stateful resource through their port-type (*implied resource pattern*);
- publish the type of distinguished properties of their stateful resources in the interface of their port-types (the *Resource Property Document*). These include properties which are specific to the semantics of the service and properties which support system-level management of the WS-Resource;
- extend their port-type to implement standard interfaces for the consumption of Resource Properties, including interfaces for state publication and discovery (*WS-ResourceProperties* [25]), for state life-time management (*WS-ResourceLifetime* [37]), and for state change subscription and notifications (*WS-BaseNotification* [24]);

Finally, many stateful services may need to recover their state after node failures or node relocations. These services ought then to persist their state, both locally (to recover from failures) and remotely (to recover from relocations).

The framework simplifies adoption and compliance with WSRF standards, in two important directions. First, it implements the interfaces and predefines properties that are common to all WS-Resources, either transparently to the developer or else by means of simple configuration. Second, it offers a comprehensive set of base abstractions for modelling stateful resources, which reflect and support common design patterns (e.g. singleton resource, multiple resources, multiple resource views over shared resources). Across all such patterns, the pre-defined abstractions factor out the basic behaviour and inter-dependencies which - regardless of service-specific semantics - may be found across the processes of creation, initialization, destruction, and persistence of stateful resources. Service-specific semantics can then be injected by extending or customizing the abstractions at the fine level of granularity that is normally associated with the use of the template pattern (callbacks).

4.7 Fault Management

Service instances throw faults to signal the occurrence of unpredictable circumstances within the runtime environment, which force deviations from the control flow expected by the implementation of the service. The intention is to characterize problems for clients in the assumption that, based on the characterization, they may react in some useful way. In particular, gCube services ought to be designed so as to return three broad types of faults, depending on whether the fault is deemed to be unavoidable across any instance of the service (unrecoverable), perhaps avoidable at some instance other than the one which observes it (retry-equivalent), or perhaps avoidable in the future at the same instance which observes it (retry-same). A resilient client may then exploit these broad semantics to react in ways which are perhaps more useful than by gracefully desisting. In particular, a client which is presented with either one of the last two types of fault can try to recover accordingly - by trying with other instances or by retrying with the same instance but at later time. Similarly, a client that is presented with an unrecoverable fault may avoid consuming further resources in the attempt.

The framework offers a number of facilities for dealing with the three types of gCube faults. First, it predefines their type declarations for immediate importing within service interfaces. Second, it provides default implementations of these interfaces which:

- serialize and de-serialize on the wire in accordance with gCube requirements;
- need not be included in stub distributions;

- can be handled as normal exceptions within the code (e.g. may wrap other exceptions and be caught in try/catch blocks).

Third, it mirrors faults with lightweight exceptions for convenience of use within the service implementation: as faults and exceptions are freely convertible, a service may be designed to:

- convert exceptions into corresponding faults as these exit the scope of the service
- convert faults received from remote services into the corresponding exceptions and let these percolate up the call stack.

While the transparencies discussed so far relate to the fulfillment of system requirements, the framework offers also tools that may support developers in the implementation of service-specific semantics, as shown below.

4.8 Discovery & Publication

Interacting with instances of other services is one of the tasks that all but the simplest gCube services and clients need to engage with. Discovering such instances by characterization of their own properties, or else those of the state they may produce and/or handle, is normally a prerequisite for the required interactions. Put another way, interacting with instances of any given service requires interacting previously with instances of services in the Information System (cf. Section 5.2).

The framework embeds a client library (more precisely the interface and the reference implementation of such library) for querying the Information Service, which simplifies the formulation of queries and the processing of results. Transparencies vary across classes of queries in reflection of different trade-offs between generality and simplicity. Free-form queries admit arbitrary query expressions and return lists of XML results wrapped inside XPath [12] engines. Named queries sacrifice the full generality of free-form queries but reduce their formulation to a simpler case of template instantiation. Resource queries specialize further to return gCube Resources (cf. Section 2.1) but offer flexibility in the specification of filters and the convenience of optimal object bindings for the results. WS-Resource queries offer similar advantages for another large class of common queries, those on WS-Resources published by stateful service instances. Finally, all types of queries can be executed by value - the execution returns with the totality of results - or by references - results are embedded in a lazily produced and lazily consumed result set (cf. Section 7.2).

The counterpart of discovery is publication and the framework offers a dedicated client library also for publishing gCube resources and WS-Resources with the Information Service. The library is used extensively and periodically by the framework itself, to publish information about the node and deployed instances, as well as by other key Enabling services (cf. Section 5). It is otherwise unlikely to be directly used by many gCube services, given that publication of gCube resources and WS-Resources which relate to individual services are carried out by the framework, either transparently to their developers or else by means of simple configuration.

4.9 Event Management

Service and client implementations that spread over a large number of components benefit from drawing loosely coupled connections between them. Asynchronous communication based on the exchange of events - possibly through third-party mediation - is a well-known approach to loosely coupled interactions which proves useful both across multiple runtimes and within single runtime.

Accordingly, the framework includes a small number of interrelated abstractions that model the prototypical actors of a local, event-based model of communication: producers accept subscriptions from consumers and notify them of the occurrence of past and future events about topics for which they subscribed. Producers, consumers, and events are parametric in the type of topics and event payloads, and can thus be instantiated more or less concretely to achieve the desired compromise between static type checking and flexibility. Focusing on producers, for example, some of them may

handle events about a given type of topic but otherwise carrying arbitrary payloads; others may allow events about any topic as long as they carry a given type of payloads; yet others may statically refuse to handler any event that is not about a given topic and carries a given payload. Events and producers are partially implemented for convenience and to ensure correctness: developers extend the former and may extend or encapsulate the latter. Topics and consumers are instead left unimplemented for flexibility: developers may enumerate topics for maximum simplicity or else organize them in arbitrarily complex hierarchies for flexibility; consumers may then be bound to individual types of topics or to roots of arbitrarily deep hierarchies. In the latter case event consumption entails a type analysis on the events in input and possibly a dispatch to event-specific methods. Producers may then simplify the task of consumers by presenting them with partial implementations that perform potentially complex type-analysis but dispatch to event-specific methods that are left unimplemented.

Similar patterns are heavily used within the framework itself to synchronize service implementations that are deployed on the same gHN and yet are unknown to each other. Through events, in particular, all service implementations synchronize transparently with key local services that are responsible for issues as critical as the management of their credentials and their lifetime.

4.10 Local Processes & Remote Interactions

From a sufficiently high level perspective, service and client implementations may be thought of as coordinating a number of local processes. Some processes may compose in chains, others may unfold in parallel, and yet others may execute periodically. Many will of course engage in interactions with remote service instances. If such processes were given dedicated object models with standard interfaces, then the most general (and tedious, error-prone, or downright complicated) aspects of their coordination - i.e. how to chain them, parallelize them, and schedule them - could be factored out and reused across implementations. Similarly, similarities between families of related processes - e.g. the steps which are common to any remote interaction, or indeed those common to all the interactions with instances of the same service - could be conveniently shared through conventional object-oriented mechanisms, i.e. by organizing implementations in inheritance hierarchies.

These observations justify one of the most versatile tools of the framework, namely a small set of interrelated abstractions used to explicitly model local processes. A handler is a stateful processor that acts upon, or on behalf of, some target object. A few key handlers are complex, in that their target is a list of 'component' handlers:

- *sequential handlers* execute their components in chains, propagating the state of each component forward and echoing failures backward;
- *parallel handlers* execute their components concurrently and apply strict or lax policies for handling their failures;
- *scheduled handlers* have a single component and repeat its execution at fixed intervals, stopping either when explicitly told or at the occurrence of some customizable condition, e.g. the amount of failures of its component.

Other pre-defined handlers specialize in best-effort interactions with remote services on behalf of the target service:

- *service handlers* query the Information Service to find service instances, engage with each of them in turn until successful, and mark instances which result in successful interactions in order to engage them first in the future. In the process, service handlers make full use of gCube fault semantics, aborting in the face of unrecoverable faults and queuing instances which are not yet ready;
- *staging handlers* extend the best-effort strategy of service handlers to the case of WS-Resources and, if none can be found in the infrastructure, try to create them through factory port-types of the service instances, again on a best-effort basis.

Developers define their own handlers against this landscape of predefined ones. For example, they may use them as the components of complex handlers with the intention

to coordinate them in some combination of sequential, parallel and scheduled execution (e.g. to parallelize two chains of processes every three minutes). Alternatively, they may extend their handlers from service and staging handlers so as to inherit their best-effort strategy and specialize it to the queries and port-types that are specific to their interactions. Families of interactions with instances of a single service could then be conveniently organized in inheritance hierarchies. Equally, they could combine complex handlers and service handlers, e.g. to poll the Information Service at regular intervals, to generate concurrently WS-resources of the same service, or to accomplish a complex task which requires a sequence of interactions which instances of different services.

5 THE GCUBE INFRASTRUCTURE ENABLING SERVICES HIGH-LEVEL DESIGN

The gCube Infrastructure Enabling Services is the family of services and related technologies called upon to support the overall operation and management of the rest of constituents of a gCube based e-Infrastructure. The facilities needed to support such kind of e-Infrastructure fall in one of the following classes: resources publishing and discovery (cf. Section 5.2), resources controlled sharing support (cf. Section 5.3), resources deployment and orchestration (cf. Section 5.4), resources selection support (cf. Section 5.5) and resource workflows definition and operation (cf. Section 5.6 and 5.7). Such facilities lead to the identification and organisation of a series of services, software libraries and related technologies described in the following overall architecture.

5.1 Overall Architecture

The overall architecture of the gCube Infrastructure Enabling Services is depicted in Figure 6. It consists of six cooperating subsystems whose role, functions and relations are briefly described in the rest of this section.

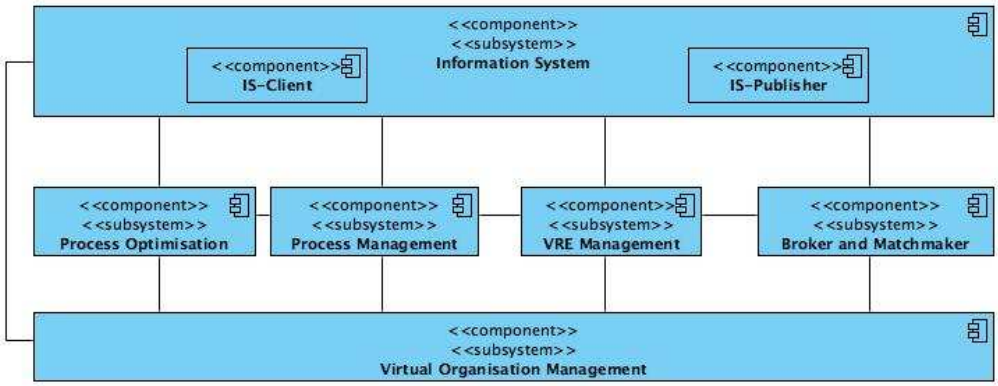

Figure 6. gCube Infrastructure Enabling Services Architecture

The ***Information System*** represents the "core" of the overall infrastructure since it is the subsystem playing the role of Registry in a gCube based Infrastructure. Because of its role, all resources partaking to a gCube based infrastructure are expected to interact with it (*i*) to *inform* the rest of the resources about its presence and its distinguishing features; and (*ii*) to discover the resources they are interested to interact with in order to accomplish its functionality. To facilitate such an interaction and to decouple the *producer* and *consumer* service logic from the internal organisation of such a subsystem two main components are envisaged, i.e. the IS-Publisher (cf. Section 5.2.4) and the IS-Client (cf. Section 5.2.6) supporting respectively the production/publishing and consumption/discovery phases in the interaction with a registry. This decoupling is fundamental since (*i*) the high work load this component is potentially subject; (*ii*) the robustness and fault-resiliency expected by such a critical component; and (*iii*) the system feature of dynamically deploying new service instances dynamically, will lead to changes in the services implementing the internals of such subsystem. This way such dynamicity is completely transparent to the Information System clients.

The ***Virtual Organisation Management*** represents the subsystem securing the sharing and reuse of the constituents a gCube based Infrastructure, i.e. all the managed resources. The subsystem implements a security framework realising the *Virtual Organisation* model described in Section 2.3 on which the D4Science Policy domain is

based. The main functions this subsystem is called upon are related to *authentication* and *authorisation*. Because of this central role, some of the components of this subsystem are expected to be ubiquitous in a gCube-based infrastructure as to facilitate the exploitation of these features. Such components are part of the gCore Framework (cf. Section 4.4).

The ***VRE Management*** represents the subsystem implementing the *Virtual Research Environment* concept (cf. Section 2.5). In particular, this subsystem supports the *definition* and *deployment* of such environments by exploiting the resources of a gCube based Infrastructure. Thus it needs to interact with the Information System to be acquainted of the resources that are available as well as of their status to select them appropriately and monitor the VRE operation. Moreover, it is requested to interact with the Virtual Organisation Management to both act securely and create the security context supporting each Virtual Research Environment. It will also interact with the Broker and Matchmaker during the deployment phase as to select the optimal pool of resources to exploit to deliver a VRE. Finally, it will interact with the Process Management in order to run some complex operations, i.e. operations involving multiple services and resources, to warm up the VRE. From an architectural point of view it is characterised by (i) services implementing the front-end (*VRE Modeler*, cf. Section 5.4.2) mediating between the user and the subsystem back-end; (ii) services coordinating the deployment and operation of the VRE (*VRE Manager*, cf. Section 5.4.3); and Services supporting the dynamic deployment (*Deployer*, *Software Repository* and *GHN Manager*, cf. Sections 0, 5.4.5, 5.4.6 respectively).

The ***Broker and Matchmaker*** represents the subsystem promoting and supporting the *optimal selection* and *usage* of resources during the VRE deployment phase. In particular, it is invoked to select the most appropriate pool of gHNs to be used, among those available in the context of the VRE, during the deployment of the services needed to operate the VRE. Because of this, it interacts with (*i*) the Information System to be aware of the available gHNs as well as of their distinguishing features (e.g. the number of Running Instances currently hosted by it, the RAM the machine is equipped with) and (*ii*) with the Virtual Organisation Management to act securely and filter out the gHNs that falls out of the operational context of the VRE. From an architectural point of view it mainly consists of a service implementing the matchmaking algorithm (cf. Section 5.5.1).

The ***Process Management*** represents the subsystem managing gCube Compound Services, i.e. complex services obtained by combining existing services, either simple or compound, in workflows delivering advanced functionality. In particular, this subsystem supports both the definition and verification of such compound services as well as their execution in a distributed scenario, i.e. the orchestration of the overall workflow is distributed among the nodes partaking to it during the execution. Because of this, it is requested to interact to both the Information System and the Virtual Organisation Management as to be acquainted of the current status of a gCube based infrastructure it is operating. From an architectural point of view, it mainly consists of (*i*) a front end supporting the user in defining such a complex workflows of services, starting their execution and monitoring their status and (*ii*) a distributed engine orchestrating the workflow execution (cf. Section 5.6.1).

The ***Process Optimisation*** represents the subsystem promoting and supporting the optimal consumption of resources during Compound Services execution. Because of this it is expected to interact with both the Information System and Virtual Organisation Management to be aware of the current status of a gCube based Infrastructure and with the Process Management Engine (cf. Section 5.6.3) to properly influence its behaviour. From an architectural point of view, it mainly consists of services dedicated to dynamically transform the execution plan governing the execution of the workflow (cf. Section 5.7.1).

The internals of each of these systems and their design principles are described in the following sections.

5.2 Information System

The Information System (IS) plays a central role in a gCube-based Infrastructure by implementing the features supporting the publishing, discovery and 'real-time' monitoring of the set of resources (cf. Section 2.1) forming a gCube-based infrastructure. It acts as the registry of the infrastructure, i.e. all the resources are registered there and every service partaking to the infrastructure must refer to it to dynamically discover the rest of Infrastructure constituents. For each resource, two kinds of information will be published:

- the *profile*, statically characterising the resource, e.g. its type;
- the *status*, characterising the operational status of the resource, e.g. indicators of the size of the resource currently managed .

Because of its central role, key requirements in terms of quality of service for such a subsystem are *performance*, *scalability*, *freshness* and *availability*. Moreover, facilities supporting the interaction with such subsystem have been included in the gCore Framework (cf. Section 4.8).

5.2.1 Reference Architecture

The functions requested to an information system fall in one of the following three phases: production/publishing, collection/storage, consumption/query. Similarly, the components forming this subsystem (presented in Figure 7) contribute to implement it with respect to one of these three functions.

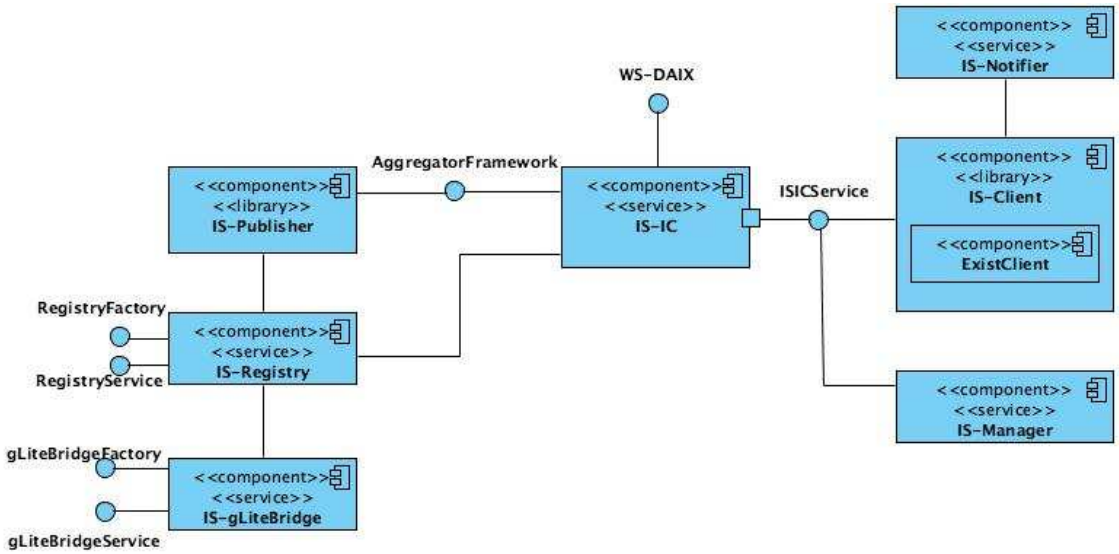

Figure 7. Information System Reference Architecture

The components supporting the production/publishing phase are:

- **IS-Registry** – this Service supports the publishing/un-publishing of *gCube resources*; a gCube resource is advertised through its *profile* (cf. Section 2.1), i.e. the resource profile represents the existence of a resource;
- **IS-gLiteBridge** – this Service supports the publishing/un-publishing of *resources* gathered from a gLite based infrastructure; a gCube-based infrastructure include resources forming a gLite-based infrastructure;
- **IS-Publisher** – this Library supports services in publishing/un-publishing groups of *resource properties* as well as registering/un-registering groups of *topics*. Actually, this library is an *interface* other Services will rely on. Because of this fundamental role in supporting Services operation in a gCube-based infrastructure, a reference implementation of such an interface (*gCubePublisher*) is part of the gCore Framework (cf. Section 4.8);

The components supporting the collection/storage phase are:

- **IS-IC** – this Service aggregates the information published in the IS; form a logical point of view it is a global registry but, because of the expected quality of service, it has been designed to support a federation model, i.e. chains of IS-IC can be configured to collectively implement the global registry function;

The components supporting the consumption/query phase are:

- **IS-Client** – this Library supports Services in retrieving information published in the IS; it supports the discovery of both *profiles* and *properties*. Actually, this library is an *interface* Services will rely on. Because of this fundamental role in supporting Services operation in a gCube-based infrastructure, a reference implementation of such an interface (*ExistLibrary*) is part of the gCore Framework (cf. Section 4.8);
- **IS-Notifier** – this Service supports other Services in subscribing/unsubscribing to *topics* produced by the various Services; this service decouples the actual producer of the topic from the actual consumer allowing for producers re-location;
- **IS-Manager** – this Service supports other Services and clients in observing, checking, or keeping a continuous record of the status of the resources forming the infrastructure. Because of this role, it can also be classified as a component supporting the collection/storage phase but it is preferable to have it in the components supporting consumption/query phase because it is considered closer to this area.

The subsystem has been conceived to rely on standards, in particular the WS-ServiceGroup [32] and WS-ResourceProperty [25] specifications. From a technical point of view it exploits the *Aggregator Framework* software framework produced by the Globus Project [23]. The Aggregator Framework is a software framework that collects data from *aggregator sources* and sends data to *aggregator sinks*. It also allows implementing pluggable and customized sources and sinks and connecting them together following the WS-ServiceGroup specification. These capabilities have been exploited in the IS by implementing some gCube services as aggregator sinks (e.g. the *IS-IC*) and allowing any gCube service to become an aggregator source (through the *gCubePublisher*); The IS-Registry is another example of aggregator source. The data exchanged within the IS connections are always *WS-ResourceProperty documents*. This allows the DIS to be as generic as possible and to be plugged with new aggregator sources at any time.

5.2.2 IS-Registry

The IS-Registry is a gCube Service called upon to manage gCube Resources by managing the registration/unregistration of their profiles. Its internals are structures as depicted in Figure 8, i.e. it mainly follows a Factory pattern [36].

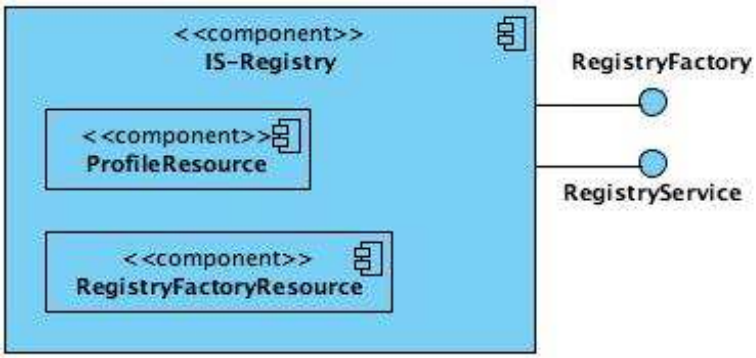

Figure 8. IS-Registry Architecture

5.2.2.1 Resources and Properties

This Service creates and manage a WS-Resource for each gCube resource that has to be considered part of the e-Infrastructure the IS-Registry is responsible for. Such WS-

Resources (*ProfileResource*) implement the Resource Model described in the Resource Domain (cf. Section 2.1), in fact each of them consists of four parts:

- A *unique identifier* identifying the resource univocally;
- A *type* characterising the class the resource belong to, e.g. Running Instance, gHN;
- A *scope* identifying the operational scope of such a resource;
- A *profile* describing the resource by resource-specific attributes.

In addition to such WS-Resources the service publishes a set of properties about its operational status including the number of resources of a certain type currently registered, the last operation performed including operation type, resource ID and time (*RegistryFactoryResource*). Such properties are also registered as Topics managed by the IS-Notifier (cf. Section 5.2.6) thus making possible for interested clients to subscribe on events representing the changes of status of Infrastructure constituents (e.g. the disappearance of a Running Instance).

5.2.2.2 Functions

The main functions supported by the Service Factory (RegistryFactory) are:

- **createResource()** – which takes as input parameter a message containing a *resource profile* and a set of *registration directives* (e.g. VO membership and VRE membership) and returns a string containing the whole profile of the new resource including the automatically assigned ID;
- **updateResource()** – which takes as input parameter a message containing the new profile that is supposed to replace an existing one. The key to identify the old profile to be replaced is contained in the profile itself, it is the resource ID. By relying on the internal mapping between IDs and EPRs it identifies the WS-Resource it has to interact with in order to implement such update operation;
- **removeResource()** – which takes as input parameter a message mainly containing the resource ID identifying the resource to be removed;
- **updateProfileWithVRE()** – which takes as input parameter a message containing the resource ID and the new VRE scopes designed to enrich the resource scope section and returns the new resource profile;
- **removeProfileWithVRE()** – which takes as input parameter a message containing the resource ID and the set of VRE scopes to be removed from the resource profile and returns the new resource profile;
- **updateState()** – which takes as input parameter a message containing the resource ID and the new state the resource has to be considered in and returns the new resource profile;

The main functions supported by the IS-Registry Service (RegistryService), i.e. the WSRF service managing the WS-Resource instances representing gCube resources, are:

- **update()** – which takes as input parameter a message containing the new resource profile and updates its current one;
- **remove()** – which deletes the resource by destroying the WS-Resource and its serialisation;
- **getProfileString()** – which returns the resource profile represented by its WS-Resource;
- **updateStatus()** – which takes as input parameter a message containing the new status the resource must appear to be and accordingly updates the WS-Resource;
- **updateVRE()** – which takes as input parameter a message containing the new VRE scopes the managed resource has to be considered in and enriches the WS-Resource accordingly;
- **removeVRE()** – which takes as input parameter a message containing the VRE scopes the managed resource has to be removed from and revises the WS-Resource accordingly.

5.2.3 IS-gLiteBridge

The IS-gLiteBridge is a gCube Service dedicated to interface with a gLite-based Grid infrastructure (e.g. EGEE), gather their resources and publish them in the gCube Information System to make them discoverable transparently like any other gCube resource. To implement such a functionality it must interface with the BDII Server, i.e. the gLite service playing the role of Information System in the gLite system. Moreover, such a service must be capable to interface with multiple infrastructures relying on gLite. The internals of the IS-gLiteBridge are structures as depicted in Figure 9, i.e. it mainly follows a Factory pattern [36].

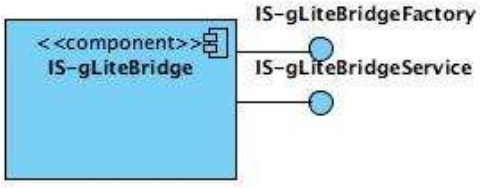

Figure 9. IS-gLiteBridge Architecture

5.2.3.1 Resources and Properties

This Service creates and manages a WS-Resource for each gLite resource that has to be considered part of the e-Infrastructure the IS-Registry is responsible for. Such WS-Resources implement the Resource Model described in the Resource Domain (cf. Section 2.1), in particular they instantiate the gLiteResource part. The following three types of WS-Resources are created and managed:

- *gLiteService* – to represent a gLite Service. It contains a *unique ID*, a *status* label and *status information*;
- *gLiteCE* – to represent a gLite Computing Element (CE). It contains a *unique ID* and information about the operational status of the resource including the number of *running jobs*, the number of *waiting jobs*, the number of *total jobs*, the *estimated response time*, the *worst response time*, the *free jobs slot*, the *max wall clock time*, the *max CPU time*, the *max total jobs*, the *max running jobs*, the *assigned job slots* and the *priority*;
- *gLiteSE* – to represent a gLite Storage Element (SE). It contains a *unique ID* and information about the operational status of the resource including the *used storage space*, the *available storage space* and the *current IO load*;

Such properties are also registered as Topics managed by the IS-Notifier (cf. Section 5.2.6) thus making possible for interested clients to subscribe on events representing the changes of status of gLite-based Infrastructure constituents (e.g. the available storage space of a SE).

5.2.3.2 Functions

The main functions supported by the Service Factory (IS-gLiteBridgeFactory) are:

- **createResource()** – which takes as input parameter a message containing the profile of the resource to be created and returns the EPR of the created WS-Resource.
- **queryForProfiles()** – which start the harvesting phase to gather resources from the BDII Server; for each resource it creates a WS-Resource of the appropriate type and interacts with the DIS-Registry to publish its profile;
- **queryForProperties()** – which starts the harvesting phase to gather resource properties from the BDII Service; for each gathered information it interacts with the relative WS-Resource and update the resource status;

The main functions supported by the IS-gLiteBridge Service (IS-gLiteBridgeService), i.e. the WSRF service managing the WS-Resource instances representing gLite resources, consist in the standard get and set operations.

5.2.4 IS-Publisher

The IS-Publisher is an interface defined in the context of the gCore Framework to decouple gCube Services from the specific implementation of the Information Service. In conjunction with the IS-Client (cf. Section 5.2.6) it represents the mediation layer gCube Services will rely on to interact with the Information Service as a whole. Such interfaces implement respectively the production/publishing (*IS-Publisher*) and the consumption/query (*IS-Client*).

5.2.4.1 Functions

The main functions supported from the IS-Client are:

- **registerGCUBEResource()** – which takes as input parameter a message containing a *resource* (*gCUBEResource*), an *operational scope* (*GCUBEScope*), i.e. an object representing a scope in the gCube model of VO/VREs, and a *security manager* (*GCUBESecurityManager*), i.e. the component keeping track of credentials, and returns a string containing the registered profile if the registration is successful;
- **registerISNotification()** – which takes as input parameter a message containing the *topic producer* (actually its EPR), a *list of topics*, an *operational scope* and a security *manager* and registers such topics in the IS-Notifier (cf. Section 5.2.7);
- **registerToISResource()** – which takes as input parameter a message containing the *topic producer* (actually its EPR), a *list of topics*, an *operational scope* and a security *manager* and registers such topics in the IS-Notifier;
- **registerWSResource()** – which takes as input parameter a message containing a stateful resource (each stateful entity different from a gCube Resource) and optionally the registration name and the scope and register such a resource in the IS;
- **removeGCUBEResource()** – which takes as input parameter a message containing a *resource ID,* a *resource type*, an *operational scope* and a *security manager* and removes the corresponding resource previously registered in the IS;
- **removeWSResource()** – which takes as input parameter a message containing a stateful resource (each stateful entity different from a gCube Resource) and optionally the registration name and the scope and removes such a resource;
- **unregisterFromISNotification()** – which takes as input parameter a message containing the *topic producer* (actually its EPR), a *list of topics*, an *operational scope* and a security manager and un-registers such topics from the IS-Notifier;
- **unregisterISNotification()** – which takes as input parameter a message containing the *topic producer* (actually its EPR), a *list of topics*, an *operational scope* and a security manager and un-registers such topics from the IS-Notifier;
- **updateGCUBEResource()** – which takes as input parameter a message containing a *resource*, an *operational scope* and a *security manager* and updates a resource previously registered in the IS.

5.2.5 IS-IC

The IS-IC is the gCube Service in charge of collecting WS-ResourceProperties (including the Resource Profiles from the IS-Registry) published through producers' components (e.g. IS-Publisher) and to make them available for consumption (cf. Section 5.2.1).

It is designed as an aggregator service, able to create Aggregator Sinks that query remote Aggregator Sources (in particular QueryAggregatorSources, as those created by the IS-IPublisher) to harvest resource properties. The collected information is stored in an embedded instance of an XML DB (eXist XML database [16]). This allows persisting the information as well as to design the IS-IC query interface by relying on the XML Query query (XQuery [8]).

From an architectural point of view, the service is composed by four WSRF Services (ISICRegistrationService, ISICEntryService, ISICFactoryService and ISICService) which globally implement the Aggregator Sink functionalities (including the ones related to the storage, indexing, and management of resource information). The ISICService also exposes the public interface to query and/or delete the stored information.

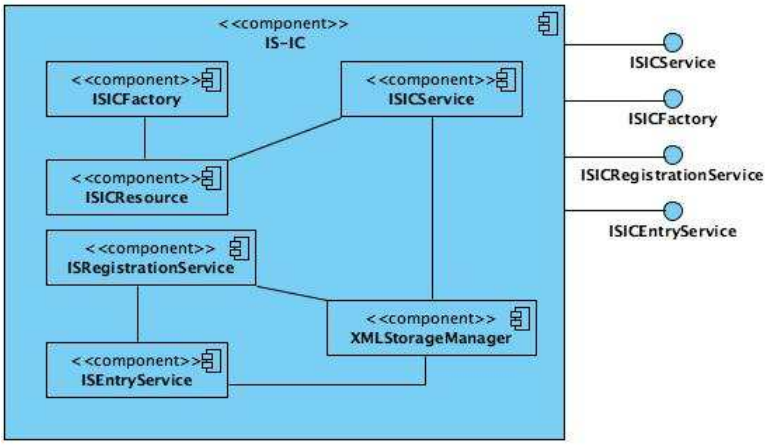

Figure 10. IS-IC Architecture

5.2.5.1 Resources and Properties

For each Aggregator Source the IS-IC is requested to aggregate an *ISICResource* is created and managed (for each Aggregator Source registration a new *ServiceGroupEntry* is created). Such a WS-Resource extends the *AggregatorServiceGroupResource* and each time the state of the ServiceGroupEntry changes the underlying Aggregator Framework invokes the relative *delivery* method.

5.2.5.2 Functions

The main functions supported by the IS-IC (through the *ISICService*) are:

- **executeXQuery()** – which takes as input parameter a message containing the XQuery to be evaluated in the context of the current information aggregated by the service and returns a result set containing an entry for each matching result;
- **deleteProfile()** – which takes as input parameter a message containing the gCube Resource ID and its type and removes the relative profile from the internal DB. This operation has to be performed by the IS-Registry only;
- **deleteResource()** – which takes as input parameter a message containing the Resource ID and removes the relative properties document from the internal DB;
- **deleteAllRPs()** – which erases all the properties stored in the internal DB;
- **dispose()** – which properly shuts down the service by closing the connections with the internal DB and terminates its running threads;
- **initialize()** – which starts up the service by opening the appropriate pool of connections with the internal DB.

5.2.6 IS-Client

The IS-Client is an interface defined in the context of the gCore Framework to decouple gCube Services from the specific implementation of the Information Service. In conjunction with the IS-Publisher (cf. Section 5.2.4) it represents the mediation layer gCube Services will rely on to interact with the Information Service as a whole. Such interfaces implement respectively the production/publishing (*IS-Publisher*) and the consumption/query (*IS-Client*).

5.2.6.1 Functions

The main functions supported from the IS-Client are:

- **getQuery()** – which takes as input parameter a message characterising the interface implementation to be used and instructing the framework to use it (by relying on the dynamic class loader mechanism);

- **execute()** – which takes as input parameter a message containing a *query*, a *query scope* and a *context* the requestor is operating in and returns the list of Information Service entries matching the query and the rest of constraints.

In addition to such functions the interface predefines the list of query typologies (a.k.a. templates) that must be implemented. Three classes of queries are envisaged:

- *gCube Resources query*, i.e. a query template to identify gCube Resources by imposing constraints on their profiles. For each existing gCube Resource a query template of this type exists, e.g. there is a GCUBERIQuery for issuing queries on Running Instance Resources, GCUBEGHNQuery for issuing queries on gHN Resources;
- *WS-Resource query*, i.e. a query template to identify properties exposed through WS-ResourceProperty;
- *generic query*, i.e. a query template to execute custom queries on the entries stored in the Information Service.

The ExistLibrary is the implementation of such a Library equipping the gCore Framework.

5.2.7 IS-Notifier

The IS-Notifier is the gCube Service dedicated to manage subscriptions with respect to *Topics*, i.e. organised items of interest for publishing/subscription [43]. It acts as a subscriptions broker since allows its client to subscribe to a topic while freeing them from the need to know the location of the notifications producer (i.e. its EPR). It also supports subscriptions to Topics that are not yet exposed by their producers. In this case, the IS-Notifier maintains a subscription in a pending state until the notification producer registers the related Topic.

It is important to notice that the design of such a service does not implement the classical brokered notification schema since the IS-Notifier subscribes the consumer directly to the notification producer. In this way, there is no single service that is in charge to receive notifications from producers and then dispatch them to the subscribed client(s). However, the IS-Notifier plays a mediator role because of the subscriptions brokerage. In fact, the IS-Notifier automatically manages the relocation of notification producers by forwarding the client subscription requests to the new instances of the producers. Moreover, the service supports the transparent subscription/re-subscription of the client to the new producer instances that comes up after the initial request of subscription.

5.2.7.1 Resources and Properties

The service adopts the singleton pattern. The managed resource represents the operational status of the service by exposing a set of properties, e.g. number of topics currently published, the number of subscriptions managed, the number of pending subscriptions.

5.2.7.2 Functions

The main functions supported by the IS-Notifier are:

- **registerTopic()** – which takes as input parameter a message containing the EPR of the topics produced and the list of *Topics* consumers can subscribe and accordingly enriches the pool of managed *Topics*;
- **unregisterTopic()** – which takes as input parameter a message containing the EPR of the topics producer and the list of *Topics* to be removed and accordingly revise the pool of managed *Topics*;
- **subscribeToTopic()** – which takes as input parameter a message containing the request for subscription (i.e. the EPR of the subscriber and the topic of interest) and subscribes the subscriber to the appropriate producer(s) according to the WS-BaseNotification specification [24].
- **listTopics()** – which returns all the Topics managed by the IS-Notifier instance;

- **listTopicForNotifiers()** – which takes as input parameter a message containing the EPR and returns ;
- **getSubscribersForTopic()** – which takes as input parameter a message containing the target *Topic* and returns the list of current subscribers (actually their EPRs);
- **removeSubscription()** – which takes as input parameter a message containing the request for un-subscription (i.e. the EPR of the subscriber and the Topic to be unsubscribed) and manages the subscription removal;

5.2.8 IS-Manager

The IS-Manager is the new service envisaged during the gCube system consolidation to replace and enhance the support previously guaranteed by the IS-Monitor [5]. The main role assigned to such a service is to equip the Information Service area with facilities for keeping track of the evolution of the information published through it along the time dimension. At the time of writing this version of the design document the service is still in its design phase and, because this phase is still non complete, the service will be described in the next version of this document.

5.2.8.1 Resources and Properties

To be defined in the next version of this design document.

5.2.8.2 Functions

To be defined in the next version of this design document.

5.3 Virtual Organisation Management

The Virtual Organisation Management (VO-Management) services are in charge to supply other gCube services with a robust and flexible security framework and to manage the VO concepts introduced in Section 2.2. The Delegation and CredentialsRenewal services provide authentication support for gCube users and services. The Authorization service is in charge to manage permissions to perform action within the infrastructure.

Authentication model

The gCube Security model is based on PKI [39] paradigm to authenticate entities identities acting in the infrastructure. This implies that each action must be performed using valid credentials issued by a trusted Certification Authority (CA) [40].

The GSI-Secure Conversation [40] standard built-in in the java-WS-Core container is used in gCube to authenticate RI invocations. In fact, there is the need to address every interaction with the system to a particular entity (user or service). For this reason all entities should have its own identity. In certain cases, services could act on behalf of a human user: GSI-SecureConversation can support credentials delegation.

This choice is driven, above all, by the need to delegate caller credentials to the invoked RI.

Authorization model

The Authorization rights in gCube are based on the RBAC model [17], and modeled as described in Section 2.3. This means that each user needs to hold a valid role to operate in a gCube-based infrastructure. The following diagram shows how the gCube VO model is implemented.

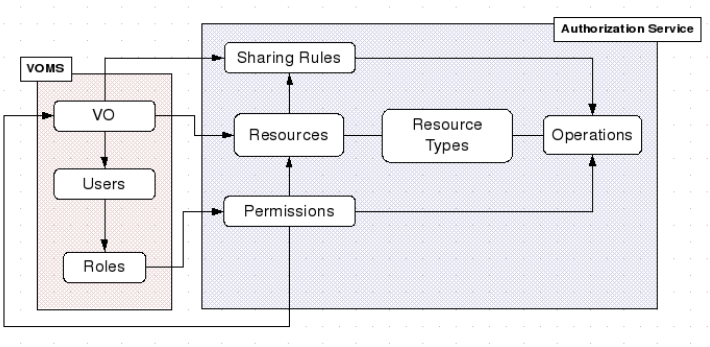

Figure 11. gCube VO Model

5.3.1 Reference Architecture

VO management is the responsibility of a set of services: the Delegation, the Credentials Renewal and the Authorization service. Precisely, the VO-management services manage the security aspects in terms of authentication and authorization mechanisms between of a gCube-based infrastructure's actors.

The VO-Management is composed of:

- **VO-Management Authorization**: (Stub library, WSRF service and API library) A service allowing VO management (VO, VOs hierarchies and gCube system VO Model of Figure 11);
- **VO-Management Delegation**: (Stub library and WSRF service) A service allowing clients to delegate proxy credentials to gCube services running on a GHN;
- **VO-Management Credential Renewal**: (Stub library, WSRF service and API library) A service allowing users to periodically delegate their credentials to GHN.

From a system wide perspective (cf. Section 3), the VO-Management services are placed in the gCube Infrastructure Enabling Services. Their main role is to support the entire infrastructure in managing authentication and authorization aspects. Referring to the VO model described in Section 2.3, VOs, Users membership, and users to roles associations are maintained by a VOMS service. In the current implementation VOs are modeled as VOMS groups, while gCube Users and Roles fits with corresponding VOMS Users and Roles.

Beyond VOMS functionalities, VO Management services offer a way to manage identities of users and services interacting with the infrastructure, through Delegation and Credential Renewal services, and a way to manage authorization rights associated to each role, through the Authorization service. The set of VO-Management services involved in providing gCube services with credentials is depicted in Figure 12. Main functionalities of the VO-Management services are described in Sections 5.3.2, 5.3.3 and 5.3.4.

5.3.1.1 Service Credentials Management

GCube system services often needs valid credentials to invoke other gCube and gLite services. These invocations can occurs either to accomplish a user request, or autonomously by the service at a given time. Sometimes the credentials delegated by the caller can be used for this purpose, but there are also cases in which the service must be provided with its own credentials. An example of this is the Hosting Node Manager (HNM) service, which periodically contacts the IS (cf. Section 5.2) to refresh profiles of services published in the local GHN. For security reasons, service credentials are not stored in the local file system, but instead periodically forwarded to services. This task, described below, is performed by the Delegation and Credentials Renewal services as shown in figure below.

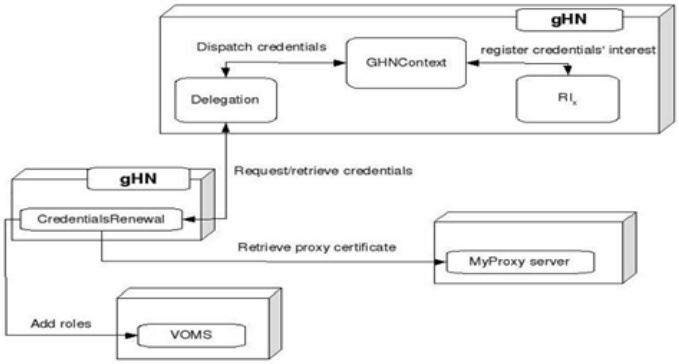

Figure 12. Authentication Management: Deployment and Interaction Diagram

So that this mechanism can work, a service responsible (not shown in picture) must delegate a copy of its credentials to the MyProxy service. Moreover, the user, or the VRE Management instance (see Section 5.4) deploying the RI (not shown in the diagram), must instruct the Credentials Renewal Service to periodically delegate these credentials. This can be done using the Credentials Renewal API library.

Once set, the Credentials Renewal Service periodically retrieves proxy credentials from the MyProxy and adds the roles needed to operate in the infrastructure, if any. Then it sends the delegated credentials to the convenient GHNContext where the RI of the service S is running. The co-hosted Delegation Service receives the delegated credentials and dispatches them to the (registered) RI of service S.

In this way, the RI of service S periodically renews the proxy credentials needed to operate in the infrastructure.

5.3.2 VO-Management Authorization

The VO-Management Authorization component provides the interfaces needed to manage a VO. It allows managing and providing persistence for the gCube Security Model. It also supports the management of VO hierarchies. The Authorization service consists of two main sub components: the Authorization Service and the Authorization API. The former is composed of a service with three WS interfaces; two of them allow to manage the VO hierarchies (following the diagram described in Section 2.3) and the other one allows to interact with a local installation of VOMS. The latter provides some helper methods to get the service port type and to get resource properties defined in the WSDL.

The VO-Management Authorization Service operates using different interfaces:

- The Authorization Service operates at root VO level. By invoking this service it is possible to manage (i.e. create, delete and modify) Operations, Roles, Resource and Resource Types, to list the VOs in the VO hierarchy and to check the access of sharing rules (given a sharing rule it controls if it is set on a VO)
- The VO Management Service operates at VO level (root or child, i.e. any VO in the hierarchy). By invoking this service it is possible to manage (i.e. create, delete and modify) VOs in the hierarchy, to set or reset a Sharing Rule in a VO and to grant or revoke Permissions from a VO.
- The VOMS Service provides an interface towards a local installation of VOMS so that it is possible to maintain updated data about users, groups and roles on the VOMS. It both wraps VOMS commands and defined customized command. This interface allows managing VOMS users, VOMS groups, VOMS roles, and their interactions.

The VO-Management Authorization API provides helper methods to retrieve the resource property VO Name and the VO Management Port Type.

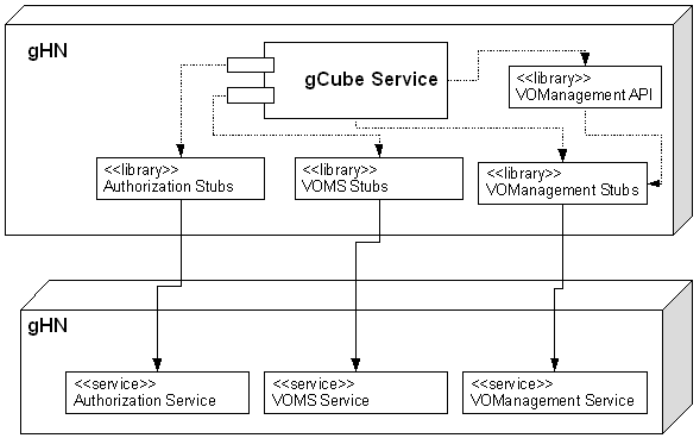

Figure 13. gCube Authorization Components

5.3.2.1 Resources and Properties

The VO-Management Authorization adopts a Factory Pattern to support VOs hierarchies and manages a Resource for each VOs Each VOResource publishes as WS-ResourceProperty the VOName of the corresponding VO. The VOResource instantiation implies, consequently, its initialization, i.e. the creation of basic Roles, Operations, Resources, Resource Types, Sharing Rules and Permissions.

5.3.2.2 Functions

The main functions supported by VO-Management Authorization are organized into three areas:

Authorization Service

- **Add/remove/modify/list Operation()** – these methods allow the management of the authorization operations;
- **create/modify/delete Role(), listExisiting/listChild/listParent Roles()** – these methods allow the management of the roles to be assigned to users and their hierarchy;
- **add/modify/removeResourceType()** – these methods allow the resourceTypes management;
- **set/resetOperationResourceType()** – these methods allow to assign/deassign operations to resourceTypes;
- **listVOs()** –returns a ListVOsResponse message with the list of the VO;
- **add/remove Resource(), listResources()** – these methods allow to add, list and remove resources to/from VOs;
- **checkAccess()** – this method allow to check accessibility based on the VOId, RoleId, OperationId, ResourceId and returns an CheckAccessResponse message with a Boolean value indicating wheter the access to the resource should be granted or denied.

VO Management Service

- **create/modify VO()** – which allow to create/modify a VOs starting from an input parameter set containing a VOID, a name and a description;
- **getParent/get Child VO()** – these methods allow to retrieve the parent/ childs of a given VO;
- **grant/revoke Permission(), listPermissions()** – these methods allow to manage the permission on the resources of the infrastructure managed by the authorization service;
- **listAvailableResources()** – to check the resources available in the VO;

- **add/remove SharingRule(), listSharingRules()** – these methods allow to manage the sharingRules on the resources of the VO managed by the authorization service;

VOMS Service

- **create/delete User(), listUserRoles(), listUserWithRole, listUserGroupRoles** – these methods are in charge of managing a user in VOMS;
- **create/delete Group(), listGroups()** – these methods are in charge of managing a group in VOMS;
- **create/delete/assign/dismiss Role()** – these methods are in charge of managing a role in VOMS;
- **add/remove Member()** – these methods allow to manage a user membership in a group;

5.3.3 VO-Management Delegation

The Delegation Service allows clients to delegate credentials to gCube services running on a GHN. Delegated credentials can then be used by co-hosted gCube services (e.g. to perform service invocations).

Each delegated credential is associated to a ServiceContext. Co-hosted services can subscribe to the local GHNContext using these identifiers to receive credentials. This can exploiting one of the gCore Framework facilities. The service is notified whenever fresh credentials for it are delegated (or cancelled) to the GHN.

5.3.3.1 Resources and Properties

The Delegation service is a singleton stateful Web Service bound to a DelegationResource resource. The DelegationResource manages delegated credentials and credential listeners subscribed by co-hosted services. For security reasons, the information managed by the DelegationResource is not persistent and it is never stored on local file system.

5.3.3.2 Functions

The main functions supported by VO-Management Delegation are:

- **delegateCredentials()** – which takes as input parameter a DelegateCredentialsInputMessage message containing the credentials to be delegated and returns an empty value.
- **cancelDelegatedCredentials()** – which takes as input parameter a cancelDelegatedCredentialsInputMessage message containing the credentials to be cancelled and returns an empty value.

5.3.4 VO-Management CredentialsRenewal

The Credentials Renewal Service allows users delegating periodically their credentials to the local Delegation Services hosted on the GHN.

The service is able to retrieve a copy of the credentials from the repository, or to create a simpleCA credentials, and forward it to the GHN interested. It can also contact VOMS server(s) adding roles needed by the services to operate.

Delegated credentials are sent to the Delegation service running on the remote GHN. Subscribed services running on that node are thus notified of the delegated credentials through the GHNContext orchestration (cf. Section 5.3.3).

5.3.4.1 Resources and Properties

The Credentials Renewal service adopts a Factory Pattern and manages CredentialsRenewalResource resource. The CredentialsRenewalResource manages credentials renewal tasks.

5.3.4.2 Functions

The main functions supported by VO-Management CredentialsRenewal are:

- **createMyProxyAccount()** – which takes as input parameter a CreateMyProxyAccountRequest message containing the username, password and contexts for the desired account and returns a CreateMyProxyAccountResponse containing an EPR of a CredentialsRenewalResource.
- **createCAAccount()** – which takes as input parameter a CreateCAAccountRequest message containing the username and contexts for the desired account and returns a CreateCAAccountResponse containing an EPR of a CredentialsRenewalResource.
- **getMyProxyAccount()** – which takes as input parameter a GetMyProxyAccountRequest message containing the username and password of a MyProxy account and returns a GetMyProxyAccountResponse containing a valid EPR of a CredentialasRenewalResource.
- **getMatchingAccounts()** – which takes as input parameter a GetMatchingAccountsRequest message containing a context and returns a GetMatchingAccountsResponse containing a matched EPR of a CredentialasRenewalResource.
- **getCredentialsAccounts()** – which takes as input parameter a GetCredentialsAccountsRequest message and returns a GetCredentialsAccountsResponse containing a list of the accounts.

5.4 VRE Management

The VRE Management subsystem is the family of services and software components responsible for the definition and the dynamic deployment of VREs. A user-friendly user interface supports VREs definitions by guidit iing the user during the specification of the desired VRE features. Through the same interface the designer is informed on the optimal deployment plan identified by the system. In fact, the resulting plan is based on availability, QoS requirements, resource inter-dependencies, and VRE sharing policies, but also on monitoring of failures (resources are dynamically redeployed) and load (resources are dynamically replicated). The VRE Manager, by coordinating three distinguished services (*Software Repository*, *Broker and Matchmaker*, *gCube Hosting Node Manager*), realises the VRE dynamic deployment by, respectively, collecting service implementations, selecting target nodes for deployment within the infrastructure, and hosting resources implementations at selected nodes.

5.4.1 Reference Architecture

As anticipated, the mission assigned to this subsystem consists in supporting the implementation of Virtual Research Environments from their definition/specification to their operation and maintenance. The services forming the subsystem have been organised according to this path as depicted in Figure 14 and described below.

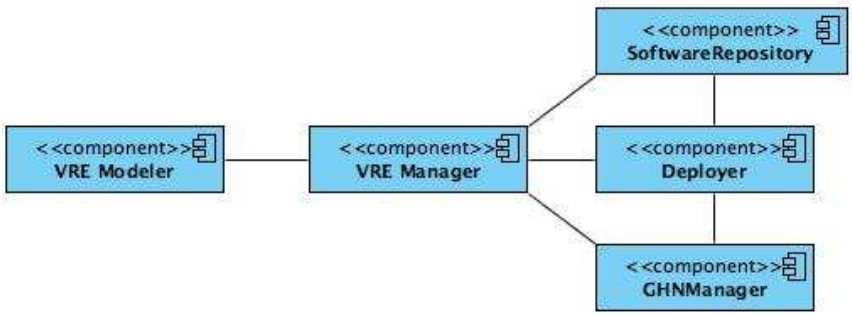

Figure 14. VRE Management Reference Architecture

To implement its role the VRE Management subsystem is organised in the following main services:

- **VRE Modeler** – this Service enables users/communities (VRE Designers) to define VREs and VRE Administrators to actually create them and monitor their operational status. It allows the definition of a set of criteria by specifying the expected characteristics of the new VRE; starting from them, it identifies the set of resources required to provide the requested features. Both the characterization criteria and the resulting set of resources constitute the ***VRE Definition document***;
- **VRE Manager** – this Service coordinates the activities needed to transform a VRE as specified in a VRE Definition document into a running set of coordinated and cooperating resources implementing it. Because of its central role, it is requested to operate with almost all services forming the VRE management subsystem as well as with the rest of the services forming the backbone of a gCube based infrastructure, i.e. the Information Service, the Virtual Organisation Management Service, the Broker and Matchmaker and the Process Management. In particular, it creates and makes publicly available through the IS the ***VRE Profile***, i.e. the operational context needed to properly operate the pool of resources forming the VRE;
- **GHN Manager** – this Service provides the *VRE Manager* service with an entry point for managing the node, e.g. installing the mandatory software, publishing node information. Bacause of this, it is expected to be deployed on each of gCube nodes as one of the local services contributing to form the gCube run-time (cf. Section 3);
- **Deployer** – this Service is another example of local service equipping any gHN in order to provide the *VRE Manager* with the facilities needed to deploy, manage and un-deploy software components on it. Through this feature, the VRE Manager can dynamically relocate the constituents of any VRE as to implement policies aiming at maximising the usage of existing assets;
- **Software Repository** – this Service is in charge of providing a gCube based infrastructure with the software components needed to operate Virtual Research Environments. In particular, it will store the software bundles that once deployed on a gHN will originate a new resource (Running Instance) that can be used in the context of one or more VREs;

5.4.2 VRE Modeler

The VRE Modeler is a gCube Service dedicated to support VRE Designers and Managers to instruct the system about the expected features of the desired Virtual Research Environment as well as allowing to easily update the environment once defined and operational and finally to drop the environment and release the exploited resource as soon as the goal the environment has been designed for is considered accomplished. Because of this (*i*) the service is expected to serve two kind of actors: *VRE Designers*, i.e. actors in charge to specify the features the VRE should provide its users with from an high-level perspective, and *VRE Manager*, i.e. actors that, with the support of the system, transforms the desiderata of the VRE Designer in term of concrete services, collections and gHNs implementing it; (*ii*) the service is expected to provide these users with its functions through an high-level User Interface.

To meet the above expectations and requirements, the VRE Modeler Service has been organized according to the Model-View-Controller architectural pattern [11] with the addition of a persistence layer. Moreover, the view and controlled layers are integrated into the same component, this is a common simplification of the original MVC framework. As a consequence, the resulting components are depicted in Figure 15.

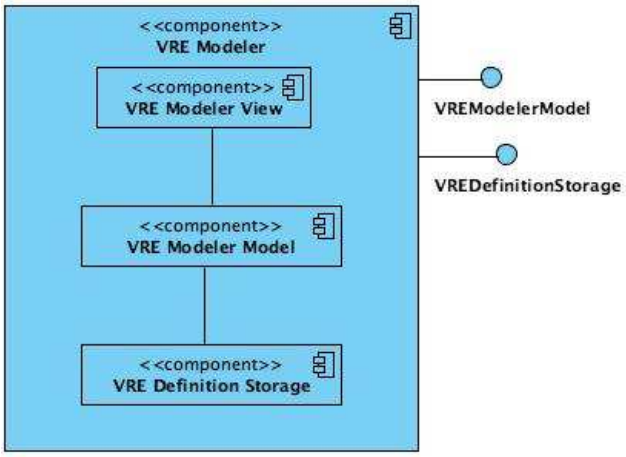

Figure 15. VRE Modeler Architecture

The ***VRE Modeler View*** is the component integrating the view and controller layers of the MVC. It is responsible for handling user interactions, performing the corresponding modification made by the VRE Designer or the VRE Manager to the VRE Definition and displaying the updated model back to the user. Moreover, it is in charge to display the operational status of the available VREs.

The ***VRE Modeler Model*** is the component implementing the model layer of the MVC. It is the functional core of the VRE Modeler Service responsible for encapsulating the appropriate data structures and exposing the procedures needed to manipulate the VRE models representing the respective VRE Definition documents. In collaboration with the VRE Modeler View, it is in charge to implement the constraints preventing the VRE Designer and VRE Manager to perform unauthorised / improper actions.

The ***VRE Definition Repository*** is the component adding persistence facilities to the MVC. It is responsible for maintaining the VRE Definition documents containing all the definition criteria of the specified VREs.

5.4.2.1 Resources and Properties

The VRE Modeler Model adopts a Factory Pattern and manages a Resource for each of the VREs it is currently handling a model. Such resource is characterised by the VRE Identifier and the specification criteria that have been specified organised in terms of *Descriptive Information*, *Content Information*, *Functionality Information*, *User Information* and *Architecture Information*.

5.4.2.2 Functions

The main functions supported by VRE Modeler Model are:

- **createVREModel()** – which creates a new VRE Model returns the reference to the resource representing it (i.e. the EPR);
- **removeVREModel()** – which takes as input parameter a message containing the VRE Identifier and removes the model representing it;
- **setDescriptiveInfo()** – which takes as input parameter a message containing the descriptive information of a VRE including its name and the creation time and updates the VRE Model accordingly;
- **getDescriptiveInfo()** – which returns the VRE descriptive information currently stored in its model;
- **setVREContent()** – which takes as input parameter a message containing the specification of the content domain of the VRE including the set of Collections to be managed and updates the VRE Model accordingly;
- **getVREContent()** – which returns the VRE content information currently stored in its model;

- **setVREFunctionality()** – which takes as input parameter a message containing the specification of the functionality domain of the VRE including the selected functions and updates the VRE Model accordingly;
- **getVREFunctionality()** – which returns the VRE functionality information currently stored in its model;
- **setVREUser()** – which takes as input parameter a message containing the specification of the user domain of the VRE including the selected users and updates the VRE Model accordingly;
- **getVREUser()** – which returns the VRE user information currently stored in its model;
- **setVREArchitecture()** – which takes as input parameter a message containing the specification of the architecture domain of the VRE including the selected gHNs and updates the VRE Model accordingly;
- **getVREArchitecture()** – which returns the VRE architecture information currently stored in its model;
- **getVREModel()** – which returns the current VRE model specification;
- **resumeVREModel()** – which initialise the VRE Model by exploiting the VRE Definition Document stored in the VRE Definition Storage;
- **deployVRE()** – which notifies the VRE Manager that a new VRE definition has been completed and the Designer asks for its deployment;

5.4.3 VRE Manager

The VRE Manager is a gCube Service grouping the facilities for managing the constituents of the VREs, namely the service instances. It is the service in charge to coordinate the overall deployment and operation of each VRE. Moreover, it takes care to interact with the Virtual Organisation Management to create the VO needed to support the operation of the VREs.

5.4.3.1 Resources and Properties

THE VRE Manager adopts the Factory pattern. In particular, it creates and manages two types of resources: the VREManagerVREScope and the VREManagerVOScope.

The former, i.e. the VREManagerVREScope, is created whenever a new VRE is created and used to keep track of the status of the VRE including the set of resources belonging to it, e.g. the list of the gHNs, the list of the Running Instances, the list of Collections. For such a resource the service exposes the VRE name and its ID.

The latter, i.e. the VREManagerVOScope, is created to manage each VO and used to manage it with respect to the purpose of this service, i.e. keep track of the set of VO mandatory services deployed. For such a resource the service exposes the VO name and the relative map, i.e. the IDs of the backbone services (non VRE specific) governing the operation of the VO and the relative VRE, e.g. the Software Repository, the IS-Registry.

5.4.3.2 Functions

The main functions supported by VREManagerFactory are:

- **createVO()** – which takes as input parameter a message containing a VO name, a Description, a Boolean value indicating whether the Virtual Organisation is requested to operate in a fully-fledged security context or not, the ID of the manager (i.e. the actor to be informed on the activities of this VO), the list of the Running Instances and gHNs that have to join the VO and returns the reference to the Resource that represents the so specified VO;
- **createVRE()** – which takes as input parameter a message containing a VRE ID, a VRE name, a Description, the ID of the designer and manager (i.e. the actor to be informed on the activities of this VRE), the URL of the resulting portal, the VRE specification (i.e. Collection, Service, Running Instances, gHNs and other constituents that have to be part of the VRE) and returns the reference to the Resource that represents the so specified VRE;

The main functions supported by VREManagerVOScopeService are:

- **DeployVO()** – which starts and coordinates the deployment actions needed to implement the given VO;
- **DeployService()** – which takes as input parameter a message containing Service ID and, optionally, the gHN ID and deploys it in the current VO. If the gHN is not specified the VRE Manager interacts with the Broker and Matchmaker to identify a suitable gHN;
- **DeploySecureService()** – which takes as input parameter a message containing Service ID and, optionally, the gHN ID and deploys it in the current VO by appropriately creating the security context for the operation of the resulting Running Instance. If the gHN is not specified the VRE Manager interacts with the Broker and Matchmaker to identify a suitable gHN;
- **UndeployService()** – which takes as input parameter a message containing the gHN ID and the Service ID and perform the un-deployment of the given service from the selected hosting node. This operations is accomplished by interacting with the gHN Manager;
- **CheckDeployStatus()** – which returns a message exposing the status of the deployment activity in the VO. In particular, the list of the currently deployed Software Archives and Packages as well as the number of re-deployment actions performed are reported;
- **CheckVOStatus()** – which returns a message exposing the current operational status of the VO in terms of the status of its constituents. In particular, this message contains the list of Running Instances currently active;
- **removeVO()** – which dismiss the VO it is invoked on. In particular, it un-deploys all the running instances serving that VO as well as all the VRE and the relative constituents running in the context of the selected VO;

The main functions supported by VREManagerVREScopeService are:

- **updateVRE()** – which takes as input parameter a message containing an updated version of the VRE specification including the list of the gHNs, the list of Services and Running Instance, the list of Collections and revises the existing VRE accordingly, e.g. by deploying new Running Instances or undeploying existing Running Instances;
- **DeployService()** – which takes as input parameter a message containing Service ID and, optionally, the gHN ID and deploys it in the current VRE. If the gHN is not specified the VRE Manager interacts with the Broker and Matchmaker to identify a suitable gHN;
- **DeploySecureService()** – which takes as input parameter a message containing Service ID and, optionally, the gHN ID and deploys it in the current VRE by appropriately creating the security context for the operation of the resulting Running Instance. If the gHN is not specified the VRE Manager interacts with the Broker and Matchmaker to identify a suitable gHN;
- **UndeployService()** – which takes as input parameter a message containing the gHN ID and the Service ID and perform the un-deployment of the given service from the selected hosting node. This operations is accomplished by interacting with the gHN Manager;
- **CheckDeployStatus()** – which returns a message exposing the status of the deployment activity in the VRE. In particular, the list of the currently deployed Software Archives and Packages as well as the number of re-deployment actions performed are reported;
- **CheckVREStatus()** – which returns a message exposing the current operational status of the VRE in terms of the status of its constituents. In particular, this message contains the list of Running Instances currently active;
- **removeVRE()** – which dismiss the VRE it is invoked on. In particular, it un-deploys all the running instances serving the VRE in exclusive manner while manages appropriately the Running Instances that serve diverse VREs.

5.4.4 Deployer

The Deployer is a gCube Service in charge to support the deployment (and un-deployment) of software packages on the managed hosting node.

5.4.4.1 Resources and Properties

The Deployer service adopts a Singleton Pattern. It manages a resource (*DeployerResource*) that exposes the operational state of the service. In particular, this resource exposes the number of software packages that have been deployed as well as their list.

5.4.4.2 Functions

The main functions supported by Deployer are:

- **deploy()** – which takes as input parameter a message containing the software components to be deployed (their identifiers) and other parameters constraining the deployment activity (e.g. the scope) and attempt to deploy them;
- **undeploy()** – which takes as input parameter a message containing the software components to be removed (their identifiers) and other parameters constraining the un-deployment activity (e.g. the scope) and drop such component from the GHN;
- **update()** – which takes as input parameter a message containing the software components to be updated (their identifiers) and other parameters constraining the update activity (e.g. the scope) and revise the software components running on the GHN;

5.4.5 Software Repository

The Software Repository is a gCube Service in charge to provide a gCube based infrastructure with the software packages that once dynamically deployed on the GHNs originates resources (Running Instances) that can be used to operate one or more Virtual Research Environments. Thus it is the repository of the gCube code that can be handled automatically by the backbone services. To implement its facilities the component will rely on Apache Maven [4], i.e. a software tool supporting the management of Java-based software projects and their artefacts.

5.4.5.1 Resources and Properties

This service adopts a Singleton Pattern. The resource managed exposes the operational status of the service by declaring the number of software packages currently managed and the number of software packages waiting for approval.

5.4.5.2 Functions

The main functions supported by Software Repository are:

- **store()** – which takes as input parameter a message containing information on the Software Archive to be stored, i.e. the Service Class, Service Name, and version identifiers and the URL the actual code is available and returns a message indicating whether the operation produced a new service or an update of a previously registered one. The new archive is stored in the repository but marked as pending, i.e. waiting for approval;
- **get()** – which takes as input parameter a message containing the Service Class, the Service Name, the Service Version, the Package Name and the version of the requested package and returns an URI clients can use to download the requested software bundle;
- **delete()** – which takes as input parameter a message containing the Software Archive Identifier and removes it from the repository;
- **listPending()** – which returns the list of Software Archive identifiers of the Software Archive stored in the Repository and waiting for approval;
- **approve()** – which takes as input parameter a message containing the Software Archive identifier and approves it as a valid component provided by this repository;

- **isDeployable() –** which takes as input parameter a message containing the Service Class, the Service Name, the Service Version, the Package Name and the version of the requested package. It returns a Boolean value indicating whether the given package is deployable (this imply that all the dependencies are deployable too) or not;
- **listScopedPackages() –** which takes as input parameter a message containing the Software Archive identifier and the required scope level and returns the list of all dependencies of the given Software Archive with respect to the specified scope;
- **listServicePackages() –** which takes as input parameter a message containing the Service Class, the Service Name, the Service Version and returns the list of packages the service is composed of.

5.4.6 GHNManager

The GHNManager is a gCube Service contributing to form the gHN (cf. Section 3). Its main role is to provide the *VRE Manager* service with an entry point for managing the node, e.g. installing the mandatory software, publishing node information;

5.4.6.1 Resources and Properties

The GHNManager adopts the singleton pattern and manages a resource (GHNResource) representing the operational context of the GHN, e.g. Running Instances hosted on it.

5.4.6.2 Functions

The main functions supported by GHNManager (thanks to the support of the gCube Application Framework and its scope management, cf. Section 4.3) are:

- **addScope()** – which takes as input parameter a message containing a new scope the GHNManager is requested to manage;
- **addRItoScope()** – which takes as input parameter a message containing a list of pairs Running Instance identifiers and scope for each of such bindings and notes this new GHN operational context;

5.5 Broker and Matchmaker

The Broker & Matchmaker (BM) service supports the VRE Management Deployer service (cf. Section 5.4.4) to identify the set of gHNs where to deploy a set of services. In particular, once the Deployer has identified the set of packages to be deployed, their requirements and their relationships, the BM's identifies the set of gHN to be used as target hosts for the deployment action.

5.5.1 Reference Architecture

The main task of the BM is the selection of the most suitable set of gHNs to deploy a specified set of packages. The matching process, implemented by the **BM-Algorithm** component, is based on the matchmaking algorithm described in section 5.5.1.1. The process returns an association between packages to be deployed and gHNs, named *deployment schema*, trying to minimize the number of gHN used. The deployment schema can then be used by the client (namely, the VREManager) to drive the deployment process.

The matching process takes into account the current status of the infrastructure, i.e. the set of services already deployed on gHNs. The **DIS-Connector** component queries the gCube Information Service (IS) to obtain the current deployment status, as well as dependencies and requirements of packages to be deployed (contained in Service Profiles and stored in the IS).

Broker and Matchmaker functionalities can be invoked by clients through the **BM-Service**. This WSRF-enabled web service, described in detail in Section 5.5.1.1, provides operations request a matching process and to notify the matchmaker of a failure in the deployment process. When a deployment process fails, the BM can be

notified about the gHN originating the failure, and the client can ask for an alternative deployment schema excluding the notified gHN.

The **BM-API** and **BM-Connector** components provide a local interface to the remote BM-Service. Particularly these components enable clients to perform asynchronous matchmaking requests. The asynchronous calling mechanism is particularly useful as the time needed to find a valid deployment schema may widely vary, depending on the status of the infrastructure (number of available gHN, number of services already deployed) and on the packages to be deployed.

A deployment diagram of BM components is shown in Figure 16.

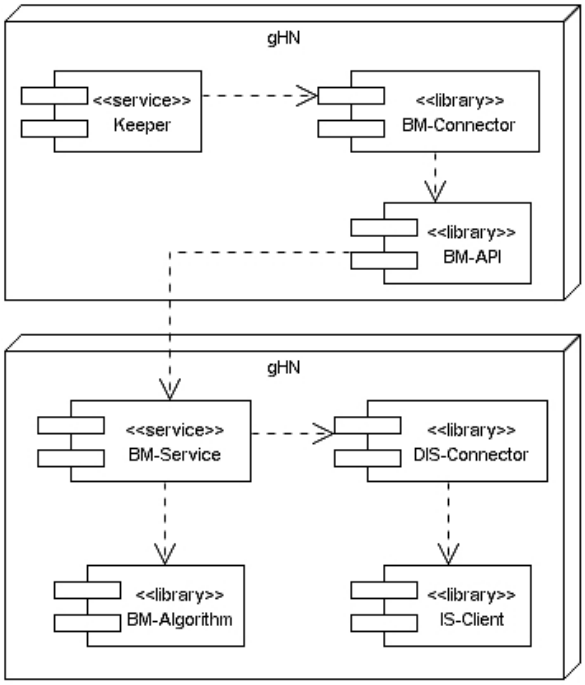

Figure 16. BM Components

5.5.1.1 Matchmaking Algorithm

The matchmaking is a variant of a Bin Packing problem [41], a combinatorial NP-hard[5] problem. In the Bin Packing a set of objects of different size must be packed into a finite number of bins of different capacities minimizing the number of bins used. The relationship between the Bin Packing and the matchmaking problem can be summarized as follow:

- Input description:
  a. A set of n items with size: d_1, d_2, ..., d_n → corresponding to packages to be deployed
  b. A set of m bins with capacity: c_1, c_2, ...,c_m → corresponding to available GHNs with their features and current status (e.g. cpu, storage, deployed services)
- Problem description:
  c. Store all the items using the smallest number of bins (i.e. GHNs)

Since the Bin Packing problem is NP-hard, the optimal solution will requires exponential time to be found. A most efficient approach uses heuristics to find a sub-optimal solution in polynomial time. Analytical and empirical evaluations suggest that the best heuristic

[5] Nondeterministic Polynomial-time hard.

for this problem is the First-Fit Decreasing algorithm. The algorithm is organized in the following steps:

Sort objects in decreasing order of size, from the biggest to the smallest

Insert each object one by one into the first bin (e.g. GHN) that has room for it. If no bin has room for it, we must start another bin

First-fit decreasing can be implemented in O(*n*logn* + *b*n*) time, where *b<=min(m,n)* is the number of bins actually used, by doing a linear sweep through the bins on each insertion.

If compared to the standard Bin Packing problem, the matchmaking algorithm has a number of additional constraints. First of all each package (item) has a set of requirements on the GHN hosting it. These requirements are of different types:

- Property requirements (e.g. OS type == Linux)
- Capacity requirements (e.g. CPU speed > 500MHz)
- Consumption requirements (e.g. disk space > 30MB)
- Software requirements (e.g. libX v2.4 has to be already installed)

Moreover, two types of dependency among couples of packages to be deployed can be specified. Supported types for relations among packages are:

- **uses** – this type of dependency implies that, after deployment, involved packages will establish some kind of interaction. Although it is desirable to minimize network communication, the fulfilment of this constraint is not to be considered mandatory for a proposed deployment solution.
- **same-host** – this type of dependency implies that the two involved packages have to be deployed on the same GHN. Dependencies of this type are mandatory; a violation would lead to a faulty deployment scheme.

These dependencies and requirements can be defined in profiles of services and further refined in a matching request.

Finally, in each request is also possible to define a list of favourite gHNs where to deploy the item defined previously.

The deployment schema resulting from the matchmaking process must take into account these requirements and dependencies and ensure that:

- For each package, its requirements are satisfied by the identified GHN;
- Each GHN satisfies the aggregation of consumption requirements of software being hosted (e.g. the sum of disk space used by packages being installed on a GHN must not exceed the current available disk space for that GHN);
- As required by the 'same-host' constraints, packages are deployed on the same hosts;
- Minimize the number of GHN hosting packages bound by a 'uses' constraint.

5.5.2 BM-Service

This stateful web service is based on the Singleton resource pattern. The BrokerResource resource, that does not publish any Resource Property, orchestrates the matching request received through the *matchOperation()* operation exposed by the service. When a request is received the BM-Service is in charge to contact the gCube IS, through the DIS-Connector, and enrich the request with: a) the current status of GHN and, b) the set of dependencies and requirements extracted from Service Profiles. Then the request is passed to the BM-Algorithm to be processed. When a response is received from the BM-Algorithm the BM-Service is in charge to format it and send back to the client. The BrokerResource is not persistent, and its status is never stored on the local file system.

As a consequence of a deployment operation, the status of the infrastructure will be modified, and the deployment scheme resulting from a subsequent matching request could be different. To avoid race conditions among concurrent matching processes the BM-Service has a sequential matching behaviour preventing two matching requests to

run in parallel. When a deployment process originated from a deployment scheme has been completed, the BM-service must be notified invoking the *notifyDeployment()* operation to start the processing of pending requests, if any. The status of the BM-Service can be queried through the *getTerminated()* operation, also exposed by the service interface.

From the security point of view the BM-Service can be invoked using the GSI Secure Conversation mechanism, and runs with the identity of the caller (credentials of the caller are delegated to the BM-service). This allows the BM-Service to impersonate the caller when querying the IS, thus obtaining just the set of GHN where the caller is authorized to deploy.

5.5.2.1 Resources and Properties

The service is a singleton stateful Web Service bound to a BrokerResource resource which doesn't publish any Resource Property. The resource also orchestrates the matching request invocation and computation. The BrokerResource is not persistent, and is never stored on local file system.

5.5.2.2 Functions

The main functions supported by the BM-Service are:

- **getTerminated()** – This operation checks if the service is available for another computation;
- **matchRequest()** – which takes as input parameter described by the BM_REQUEST XML Schema, invokes the matchMakerAlgorithm method of the BM Algorithm component and returns its result described by the BM_RESPONSE XML Schema;
- **notifyDeployment()** – It allows sending a notification about the outcomes of the deployment activities perfomed by the caller.

5.5.3 BM-API & BM-Connector

The BM-Connector and BM-API are Java Libraries used to contact the BM-Service. The BM-Connector component provides a notification mechanism based on the *Observer-Observable* Design Pattern [19]. The *BmmConnector* class allows clients to submit matching requests and deployment feedbacks. The *BmmObserver* interface must be implemented by clients to receive notifications about completion of matching processes. This interface is also used by the *BmmConnector* object to notify the reception of a deployment feedback.

5.5.3.1 Functions

The main functions supported by the BM-API and BM-Connector libraries are:

- **submitRequest()** – which takes as input parameter a message described by the BM_REQUEST XML Schema, invokes the matchMakerAlgorithm method of the BM Algorithm component and returns its result described by the BM_RESPONSE XML Schema;
- **notifyDeployment()** – It allows sending a notification about the outcomes of the deployment activities perfomed by the caller.

5.6 Process Management

The Process Management class of services is dedicated to the management of gCube Compound Services (CS), i.e. complex services obtained by composing other simpler services through an appropriate workflow management language. The components in this family cover all aspects related to CS management. In particular, they allow to design via a dedicated user interface the CS and perform various validation activities related to CS design, to manage CS definitions, to execute instances of the services and control in various ways the execution. Finally, a component in this family allows managing gLite jobs as "services" in a CS.

5.6.1 Reference Architecture

Here we give an overview of the functionality offered by this class of services as a whole, while the individual services and their interfaces are described in more detail in the next section. The last part of this section is dedicated to various architectural considerations regarding the components dedicated to designing processes, with which the end user is supposed to interact. The functionality of the Process Management Class can be grouped from a logical point of into various areas of competence as depicted in Figure 17.

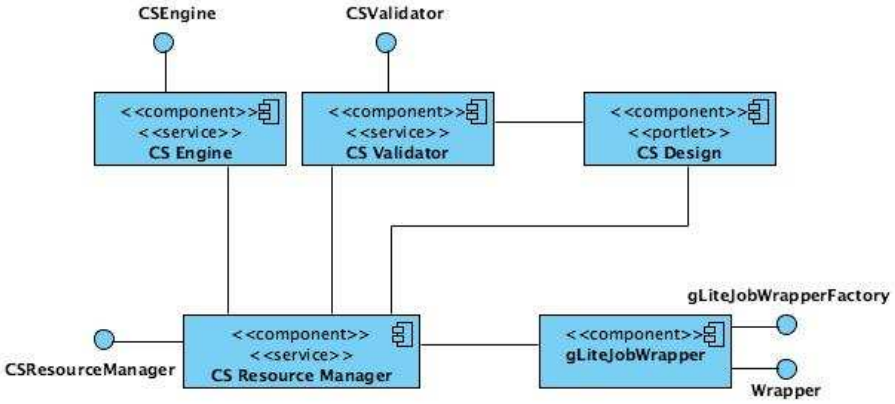

Figure 17. Process Management Reference Architecture

The components are: (*i*) The **Process Design and Monitoring Portlet**, responsible for providing a user interface for viewing, editing and managing compound service definitions in a gCube based infrastructure, and for triggering the execution of existing/new processes; (*ii*) the ***CSValidator*** service, responsible for the validation of the new processes; (*iii*) the ***CSEngine*** service, dedicated to the orchestration of the distributed execution of a compound service; (*iv*) the ***CSResourceManager*** service, which provides common operations required by other Process Management components, and in particular operations for handling (registering, un-registering) process resources, i.e. Process specifications and process instances, on the gCube Information System and to issue complex process-related queries against the gCube Information System; and (*v*) the **gLiteJobWrapper** service, that allows to invoke gLite jobs in the same way as any other gCube service, and therefore provides the means to integrate them into the (service-oriented) workflows that the process management class of service handles.

These components are all described next, with the exception of the Process Design and Monitoring Portlet, which is described in Section 8.4.11.

5.6.2 CSValidator Service

This service represents the point of reference for all activity related to validation. It performs syntax checks, semantic checks, and, of course, checks any transactional or other additional properties the process specification may have. The process validation is a necessary step if the process needs to be later executed and involves syntactical and semantic correctness checks. Optionally, it can also check any transactional or other additional properties that the process specification may have. The result of the validation process is a "validation signature" confirming that the process is correctly defined. This is a required step since the runtime engine will reject processes without the validation signature or those whose signature is corrupted.

The implementation of the CSValidator service is based on the Apache Project's XML Security Java libraries.

5.6.2.1 Resources and Properties

The CSValidator service is implemented as a stateless WRSF-compliant service. It does not publish any resource on the Information system.

5.6.2.2 Functions

The main functions supported by the CSValidator are:

- **validate()** – which takes as input parameter a message containing a process specification and a process schema against which the process must be validated and returns a signed process specification if the specification if valid;

5.6.3 CS Engine Service

The CSEngine is the core service used for the orchestration of the distributed execution of a compound service. It is responsible for starting a process and executing process activities, and for routing the process execution to the next node.

The CSEngine is based on a distributed process execution engine called OSIRIS. The functionality exposed by the service is in part implemented inside an internal library, the OSIRIS library. The library is meant for internal use by the CSEngine only and it's not described here in detail. OSIRIS supports a custom graph based process description model, which can be partially mapped on the BPEL process description language. The main constructs of the model are the following activities:

- **SOAPCall** - this activity provides the functionality to send a given SOAP message to a web service defined by its endpoint and optionally its SOAP action (as defined in the WSDL for the operation). It corresponds more or less to the BPEL "invoke" activity. It takes as input arguments the EPR of the service to be invoked, the operation to be executed, the message to be passed to the service operation, a flag indicating alternative calling mechanisms, the number and list of additional headers to be passed in the SOAP call and returns the response message resulting from the service invocation;
- **Assignment** –this activity, which corresponds to the BPEL "assign" activity, provides the functionality to assign any number of process variables (or parts thereof) or expressions to other variables or parts thereof. It takes as input parameter the list of source of assignment and returns the list of results of the requested assignments;
- **Wait** – this activity, which corresponds to the BPEL "wait" activity, provides the functionality to wait for a specified amount of time, i.e. to pause process execution. Both types of parameters that are mentioned in the BPEL specification are supported, and exactly one of "For" or "Until" must be given. It takes as input parameters the specification of the time to wait.
- **Empty** – this activity corresponds to the BPEL "empty" activity. The semantics are those of a no-op, i.e. the activity does nothing and has no effect.
- **Switch** – this activity corresponds to the BPEL "switch" activity. i.e. its semantics are those of a conditional branch.
- **While** – this activity corresponds to the BPEL "while" activity, i.e. its semantics are those of a loop with an exit condition. OSIRIS process definitions are only allowed in the form of while activities.

5.6.3.1 Resources and Properties

The CSEngine is a stateful web service. Each running CSEngine service provides a WSRF resource for process instances it is involved in. The purpose of these resources is to allow for the monitoring (cf. Section 8.4.11) of the execution of a process, as the execution proceeds, for the coordination of process joins (where parts of the process might be executed on different nodes of the infrastructure and need to "agree" on the node where the process join takes place), and for retrieving the result of a process which was run asynchronously.

All resource properties that are available for the process instance have a dynamic QName (based on the process instance ID, and possibly information on which step in the

process they refer to) for accessing them and are created at runtime. In addition, all resource properties are made available as WS-Notification Topics and registered in the IS-Notifier (cf. Section 5.2.7). The following list provides an overview of the different Resource Properties a CSEngine may expose. The placeholder <instanceid> corresponds to an actual UUID of the corresponding process instance:

- status_<instanceid> holds status information about the process instance. Valid values are "Running", "Finished", "Finished with error", "Abort requested", and "Aborted";
- log_<instanceid> holds log information about the activities performed in the process. This resource property holds an array of Strings corresponding to the different log entries;
- routing_<instanceid> holds information about CSEngine instances that the process execution has been routed to;
- activity_<instanceid> - this resource property is created when the current node is responsible for coordinating a process join for the activity <activity>. CSEngines executing other branches that need to join at the given activity subscribe to and read this resource property to be informed where the process join should take place;
- involved-nodes_<instanceid> is used merely as a flag for signalling that this node was involved in the execution of the instance. The corresponding entry contains an array of Strings with the output of System.currentTimeMillis(), indicating when the process was handed over to this node;
- outputavailable_<instanceid> is used for signalling that process output is available.

5.6.3.2 Functions

The main functions supported by the CSEngine are:

- **startCS()** – which takes as input parameter a message containing the process ID, a Boolean indicating whether the process has to be executed asynchronously and a list of parameters and returns (i) the process instance ID of the created instance if the process has to be executed asynchronously, or (ii) the list of the results resulting from the execution of the process if the process has to be executed synchronously;
- **startAdHocCS()** – which takes as input parameter a message containing a serialised version of an entire *process definition*, a Boolean indicating whether the process has to be executed asynchronously and a list of parameters and returns (i) the process instance ID of the created instance if the process has to be executed asynchronously, or (ii) the list of the results resulting from the execution of the process if the process has to be executed synchronously;
- **getGenericCSOutput()** – which returns the list of the results resulting from the execution of the process represented by the Resource it is invoked on;
- **cleanupInstances()** – which cleans up the resources created for process instances from the local CSEngine it is invoked in. Two flags on the input message may indicate to also propagate the cleanup to the other CSEngines and to remove the corresponding GCUBECSInstance(s) from the IS.

5.6.4 CSResource Manager Service

The CSResourceManager Service provides operations for storing (registering) and removing Process definitions and Running Process Instances, thereby acting as a proxy to the resource-related functionality of the gCube Information System. It also provides an abstraction layer for queries directed to the IS, in order to have a simple and tailored interface which can be used from all of the Process Management components. While all the "depending" components could, in theory, use the IS Client directly, the same (complicated) queries would have to be used in multiple places; therefore, the decision was basically to outsource the queries into a common location where they can be used in a uniform way, and duplication of effort is avoided.

5.6.4.1 Resources and Properties

The service is designed as a stateless WSRF-compliant web service. It does not publish any resource.

5.6.4.2 Functions

The operations provided by the Resource Manager Service are listed below. All operations are implemented as static methods, called "lib<...>", e.g. "libSaveSpecs". This allows for also directly invoking the Java methods locally (from services installed on the same DHN as the PRS), bypassing the overhead otherwise introduced by the GT4 infrastructure. All operations which are not static ("<...>", e.g. "saveSpecs") are available as WS operations and are merely wrapping the static methods.

- **libSaveSpec()** – which takes as input parameter a message containing a process specification and returns the Resource ID resulting from the publishing of such a resource in the IS;
- **saveSpec()** – the WS version of the above function;
- **libSaveSpecs()** – which takes as input parameter a message containing a list of process specification and returns the list of Resource ID resulting from the publishing of such resources in the IS;
- **saveSpecs()** – the WS version of the above function;
- **libSaveInstance()** – which takes as input parameter a message containing the process ID of the running instance it is running and returns the process instance ID resulting from the publishing of such resources in the IS;
- **saveInstance()** – the WS version of the above function;
- **libUpdateInstance()** – which takes as input parameter a message containing the process ID of the running instance it is running, the profile running instance ID and the new profile and returns the process running instance ID resulting from the publishing of such resources in the IS;
- **updateInstance()** – the WS version of the above function;
- **libRemoveSpec()** – which takes as input parameter a message containing the process ID and returns a Boolean value indicating whether the process has been successfully un-registered from the IS or not;
- **removeSpec()** – the WS version of the above function;
- **libRemoveInstance()** – which takes as input parameter a message containing the process running instance ID and returns a Boolean value indicating whether the process has been successfully un-registered from the IS or not;
- **removeInstance()** – the WS version of the above function;
- **libGetProcessDefinition()** – which takes as input parameter a message containing the process ID and returns the process definition of such a process;
- **getProcessDefinition()** – the WS version of the above function;
- **libSearchForProcessDefinitionsByRegex()** – which takes as input parameter a message containing a processing filtering condition and returns the list of process definition registered in the IS that matches the specified criteria;
- **searchForProcessDefinitionsByRegex()** – the WS version of the above function;
- **libGetServicesOnGHN()** – which takes as input parameter a message containing a GHN ID and returns the list of running instances currently hosted on the specified GHN by providing the GHN ID, the porttype univocally identifying the service, and the EPR the running instance can be reached;
- **getServicesOnGHN()** – the WS version of the above function;
- **libGetCSInstances()** – which takes as input parameter a message containing a process ID, the status of the running instance and returns the list of the current process running instances matching such criteria by providing the process ID, the process running instance ID and the operational status;
- **getCSInstances()** – the WS version of the above function;
- **libGetRunningGCubeInstances()** – which takes as input parameter a message containing a porttype univocally identifying a gCube service and, optionally, a gHN ID

and returns an array of running instances matching such criteria by providing the gHN ID, the porttype and the EPR;
- **getRunningGCubeInstances ()** – the WS version of the above function;
- **libSearchForGCubeServicesByAnyText()** – which takes as input parameter a message containing search criteria (specifying descriptive aspects like service class, service name, package name, entry name, porttype namespace and local name) and returns the list of the gCube service resources matching such criteria by providing the the service class, the service name, the package name, the entry name and the porttype;
- **searchForGCubeServicesByAnyText ()** – the WS version of the above function;
- **libSearchForGCubeServicesByFilters()** – which takes as input parameter a message containing search criteria (specifying descriptive aspects like service class, service name, package name, entry name, porttype namespace and local name) and returns the list of the gCube service resources matching these criteria (logical AND) by providing the the service class, the service name, the package name, the entry name and the porttype;
- **searchForGCubeServicesByFilters()** – the WS version of the above function;
- **libGetGHNInfo()** – which takes as input parameter a message containing a GHN ID and returns the list of existing GHNs by providing their ID, hostname, IP address, the site name, the site country and other information maintained in the GHN profile like the RAM size;
- **getGHNInfo()** – the WS version of the above function;
- **isEndpointReachable()** – which takes as input parameter a message containing the EPR of a Running Instance and returns a Boolean value indicating whether the running instance has to be considered reachable or not by trying to establish a socket connection.

5.6.5 gLite Job Wrapper

The GLiteJobWrapperService aims at providing a service-oriented interface to gLite jobs. The jobs are sent to the gLite infrastructure using a specified WMProxy (a webservice-based access point to gLite job submission).

5.6.5.1 Resources and Properties

It is a stateful WSRF service following the Factory/Instance pattern. It uses WS-Resources to represent the gLite jobs. Such a kind of resource contains the following job information:
- *JDL*: the Job specification file for the job; This JDL is directly passed on to the underlying gLite system for executing the job;
- *Job Id*: the ID assigned to the job once it has been submitted to the gLite infrastructure;
- *Input and output sandbox files*: complex objects containing the input sandbox files for the job and the output sandbox files as retrieved after the job has finished, respectively. Only meaningful operations to act on these files are provided to the user, i.e. the input sandbox is only writeable, whereas the output sandbox is only readable by the user.

5.6.5.2 Functions

The main functions supported by the gLite Job Wrapper service are:
- **createResource()** – which returns the EPR of a new resource created to manage a job;
- **setJDL()** – which takes as input parameter a message containing a serialisation of the JDL specification of the job and assign it to the resource it is invoked on;
- **setInputSandboxFiles()** – which takes as input parameter a message containing a list of sandbox files by specifying their name and binary content and assigns it to the resource it is invoked on;

- **submitAndWait()** – which submits the job represented by the resource it is invoked on, assign the job ID to the resource and waits for its execution to be finished;
- **getOutputSandboxFiles()** – which returns the list of sandbox files, by providing their names and binary content, resulting from the execution of the job represented by the resource it is invoked on;

5.7 Process Optimisation

The gCube Process Optimisation Services (POS) implement core functionality in the form of libraries and web services for optimised Process scheduling and execution planning. POS comprises a core optimisation library (POSLib) and two Web Services (RewriterService and PlannerService) that expose part of the library's functionality. POSLib implements three core components of process optimisation. POS is an integral part of query execution in gCube, since it is responsible for the optimised scheduling of workflows produced by the query planner and consequently is a key player in alleviating the Grid overhead in query execution.

gCube Process Optimisation components are involved in two distinct cases by the Process Execution Service (PES):

- Just before the execution of a Compound Service: the workflow definition is passed to POS so as to process it for enhancing it in terms of parallelization and resource usage;
- Just before a particular invocation, upon need to (e.g. failure of some node) so as to identify a new candidate for the invocation (a process called "Active Planning").

5.7.1 Reference Architecture

The following figure provides an abstract overview of the Process Optimisation Services design focusing on the key components comprising the subsystem. Detailed description of each component is provided in the following subsections.

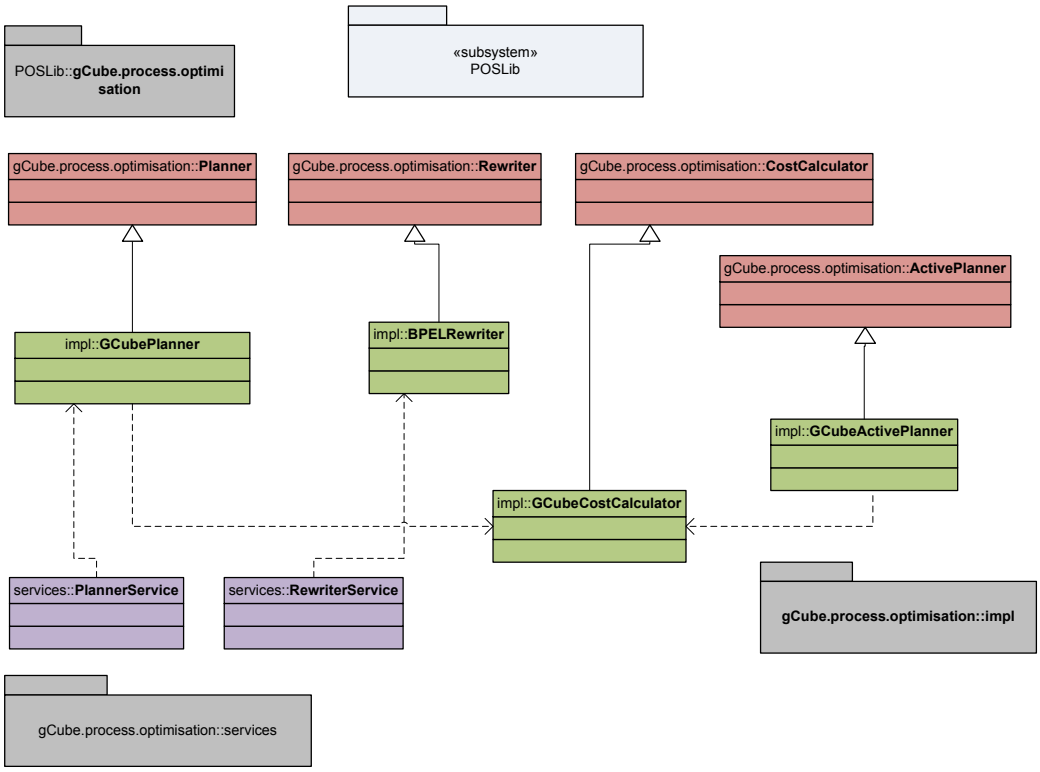

Figure 18. POS Internal Components

5.7.2 Planner Service

Performs the pre-planning of the process execution. Receives an abstract BPEL process and generates various scheduling plans for execution. The generation of an executable plan implies that all references to abstract services are replaced by invocations to concrete, instantiated services in a gCube infrastructure. The Planner uses information provided by the Information Service which keeps up-to-date metrics for resources employed in the Grid (physical machines, services, etc). This information is input to various cost functions (applied by the paired Cost Estimator) that calculate the individual execution cost of a candidate plan.

The selection of best plans is performed by a custom implementation of the Simulated Annealing algorithm. The outcome of the planning is a set of executable BPEL processes that are passed to the workflow execution engine. Cost calculation can be guided by various weighted optimisation policies passed by the author (human or software) of the BPEL process inside the BPEL description.

The Planner class is concretely implemented in the context of gCube by the *GCubePlanner*. The latter is integral part of the POSLib library and has also a Web Service frontend provided by the *PlannerService*, which is a regular gCore service.

5.7.2.1 Resources and Properties

The Planner Service is a stateless service.

5.7.2.2 Functions

- **createPlan()** receives a BPEL Document defining the compound service workflow and returnsa list of execution plans sorted by their execution cost.

5.7.3 Rewriter Service

Provides structural optimisation of a process. It receives as input a BPEL process, analyses the structure, identifies independent invocations and formulates them in parallel constructs (BPEL *flow* elements) in order to accelerate the overall process execution. It is the first step of optimisation that takes place before the process arrives at the execution engine. As with the Planner, the Rewriter offers a gCore Web Service frontend in the form of RewriterService. The latter is actually a wrapper for the concrete BPELPlanner class, which implements the abstract Rewriter class.

5.7.3.1 Resources and Properties

The rewriter service is a stateless service.

5.7.3.2 Functions

- **optimiseProcess()**: receives the BPEL document of the process and produces BPEL document with the restructured optimised process workflow

5.7.4 POSLib

POSLib provides a set of classes that support the operation of the POS Services. Apart from the classes already mentioned, POS Lib provides additionally the following elements:

- *ActivePlanner* provides run-time optimised scheduling of a gCube process. It is invoked during the process execution before any invocation activity to ensure that the plan generated by the Planner (during pre-planning) is still valid (e.g. the selected service end-point is still reachable) and optimal (according to the user-defined optimisation policies). If any of the former criteria has been violated the ActivePlanner re-evaluates a optimal service instance for the current process invocation. It can also work without pre-planning being available. The abstract ActivePlanner class is implemented in the context of gCube framework by the GCubeActivePlanner class. The latter is part of the POSLib library.
- The *CostCalculator* which is the backbone of the optimisation functionality. It implements the necessary logic to quantify the execution semantics of the activities in order to facilitate the comparison between different but equivalent plans and the selection of the "best" activities among the candidate set. It is an essential component for the Planner and the ActivePlanner. In the context of gCube the component is implemented by the GCubeCostCalculator class. The functionality of the CostCalculator is not exposed directly to any other gCube components. Nevertheless, much of the behaviour and attributes of POS subsystem relies on the cost calculation strategies implemented within this component.

Among the exceptional features of the POSLib is the implementation of a Simulated Annealing algorithm, as a private method of a GenericPlanner object, for identifying the optimal plan. Other design considerations are presented in section 5.7.5.

5.7.4.1 Functions

Main functions offered by the library are:

- **calculatePlanCost()**: which calculates the cost of the execution of a plan according to a policy.
- **createPlan()**: which weceives a compound service and produces a list of execution plans by assigning concrete running service instances to the process graph nodes.
- **suggestNext()/suggestNextList():** which evaluates a list of candidate Running Instances for the next step of process execution.
- **costFunctionXXXX()**: A set of functions which receive an item of appropriate type (e.g. service class) and returns the cost of the given service based on the information provided by the service profile in the IS and the metric XXXX.

5.7.5 Design Considerations

A number of aspects of the design of POS are presented bellow with regards to different aspects of its expected operation.

5.7.5.1 Optimisation Policies

The Planner and ActivePlanner components perform optimised scheduling of abstract BPEL processes based on user-defined policies. These policies guide the CostCalculator during the quantification of planner execution. Optimisation policies are declared within the BPEL document and can apply to individual *partnerLinkTypes* or to the whole process.

In BPEL, *partnerLinkTypes* define the classes of Web Services that can participate with multiple roles in a process. A particular instantiation of a *partnerLinkType* inside the process is denoted by a *partnerLink* element definition. A process may include various different parterLinks from the same *partnerLinkType* participating in the process with different roles. In practice a *partnerLink* is the Web Service whose operations can be invoked during process execution.

The selection of a specific Web Service instance to be used in a particular process invocation is driven by the optimisation policy applied either on the process level or on a *partnerLink* level. Currently POS supports six different optimisation policies:

- **Host load**: Hosts with the lowest system load take precedence.
- **Fastest CPU**: Hosts are ranked according to their CPU capabilities and the best is selected.
- **Memory Utilisation**: Hosts are ranked according to the percentage of available memory as reported by the Java VM. The one with the highest percentage is selected.
- **Storage Utilisation**: Hosts are ranked according to their total available disk space. The one with the larger available space is preferred.
- **Reliability**: Hosts are ranked based on their total uptime. Precedence is given to hosts that have been running without interruption for longer time.
- **Network Utilisation**: When the Planner evaluates multiple possible scheduling plans it will show preference to those plans where the web services are located close to each other (based on the reported host locality information). The Planner will avoid co-scheduling invocations to the same host in order not to overload it.

In order to perform the various cost estimations for the one or combination of two or more policies, the CostCalculator collects various metrics from the gCube information system (IS) via the ServiceQueryBridge component. These metrics reflect the state of the VRE and constituting

5.7.5.2 Abstract and Concrete Service References

In gCube we distinguish three categories of partnerLinkTypes:

- **concreteGCubeService**: Is a static reference to a specific Running Instance. The Planner will not try to reschedule the assignment of this partnerLinkType
- **concreteExternalService**: Similar to the previous but points to a Web Service outside a gCube infrastructure.
- **abstractGCubeService**: partnerLinkTypes whose partnerLinks can be rescheduled at any time either by the Planner or the ActivePlanner. The selection of the specific Running Instance to use depends on the optimisation policies declared in the BPEL document and the state of the VRE infrastructure.

5.7.5.3 BPEL Optimisation Extensions

gCube POS functionality heavily depends on the BPEL standard (notation based on BPEL4WS v1.1). BPEL XML schema has been extended to include optimisation information such as process policy information per partnerLinks, the definition of abstract or concrete services, allocation relationship between invocations etc.

To define process wide optimisation policies we introduce the *optimisationPolicy* attribute at the BPEL process element. For example the process defined by the BPEL excerpt in Figure 19 will be scheduled according to the *fastest_cpu* policy (with higher weight) and the *storage_utilization* policy (lower weight). If no policy is defined the default used is the *host_load* policy.

```
<process optimisationPolicy="fastest_cpu storage_utilization" xmlns:... >
    <partnerLinkTypes>
        <partnerLinkType name="BPELD4SProcess">
            <role name="BPELD4SProcessProvider">
                <portType serviceType="concreteGCubeService" name="tns:BPELD
            </role>
        </partnerLinkType>
        <partnerLinkType name="fulltextindexlookupserviceLT">
            <role name="fulltextindexlookupserviceRole">
                <portType xmlns:fulltextindexlookupservice="http://diligentp
            </role>
        </partnerLinkType>
        <partnerLinkType name="sortoperatorserviceLT">
            <role name="sortoperatorserviceRole">
                <portType xmlns:sortoperatorservice="http://diligentproject.
```

Figure 19. Process-wide Policy Definition and Definition of *serviceType* per *partnerLinkType*

The policies defined on process-wide level pertain the planning of all partnerLinks included in the process unless a specific policy is defined on the partnerLink element. To define such policy one uses the partnerLinkPolicyType attribute of the BPEL partnerLink element. This attribute is used similarly to the above example, except that the *network_utilization* policy, if defined, is ignored, since this policy makes sense only for the whole process and not for a particular web service.

6 THE INFORMATION ORGANISATION SERVICES HIGH-LEVEL DESIGN

The *gCube Information Organisation Services* is the family of subsystems implementing the services supporting the management (storage, organisation, description and annotation) of information. These services implement the notion of *Information Objects*, i.e. logical unit of information potentially consisting of and linked to other Information Objects as to form *compound objects*. in the services are organised in three main functional areas: (*i*) the storage and organisation of such Information Objects and their constituents (*Content and Storage Management*); (*ii*) the management of the metadata objects (actually implemented as a kind of Information Object) potentially equipping each Information Object (*Metadata Management*); and (*iii*) the management of the annotations objects (actually implemented as a kind of Information Object) potentially enriching each Information Objects (*Annotation Management*).

6.1 The gCube Content Model

While other infrastructures for the manipulation of content in Grid-based environments, like gLite, provide basic file-system like functionality for content manipulation, the Information Organization services are aimed to provide more high-level functionality, built on top of gLite or other storage facilities. Content is stored and organized following a graph-based data model, the *Information Object Model*, that allows finer control of content w.r.t. a file based view, by incorporating the possibility to annotate content with arbitrary properties and to relate different content unities via arbitrary relationships.

Building on this basic data model, other services in the Information Organization family provide to other gCube services more sophisticated data models to manage *complex documents*, document *collections, metadata* and *annotations.*

6.1.1 Information Object Model

The elementary constructs of the model are **information-objects** (a node of the graph) and **object references** (the arcs). The ER Diagram in Figure 20 describes the model.

- An Information Object (IO) represents an elementary information unity. It is uniquely identified by an Object Identifier (OID), is labelled with a *name*[6] and a *type*[7] and Information optionally annotated with a number of properties. These properties are simple key-type-value associations. Finally, it can be associated with a raw-content. The raw content of an object is content of any kind. The model hides the actual storage details of the content of an object, that can be for instance stored as a file in gLite or as BLOB-field in a database, or maintained in storage facilities not under direct control of the Information Organization Services, e.g. as file stored in a remote server and accessible through some protocol like http, ftp or gridftp.
- An object reference "links" two Information Objects. Each object might (*i*) reference many other objects and (*ii*) be referenced by many objects (m-n relationship). A reference is directed, it is labelled with a type attribute, called *primary role*, a *secondary role*, that may optionally further specify the function of the primary role[8],

[6] The 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, and can be empty.

[7] The type identifies the kind of Information Object, like document, collection, aggregate, metadata, external document etc. It is a mandatory attribute.

[8] Examples of relation types (in terms of *primary role*:*secondary role*) currently supported are *contentmanagement:is-member-of,* indicating that the source object is a member of the collection target Information Object; *contentmanagement:is-described-by* to indicate that the target takes the role of metadata for the source object and therefore describes the source; *contentmanagement:has-part* to indicate that the target document/collection is a part of the source object; *contentmanagement:is- represented-by* to indicate

and a position attribute, that allows to build ordered graph structures. It can also be associate with a number of other properties.

The information-object model introduced above is exposed to higher level Information Organization Services by a component called the Storage Management Service (cf. Section 6.3). The generality of this simple information model allows to build complex data-structures. The services within the Information Organization stack build on top of this model to offer an organized, high-level view of content. This is done by attaching specialised semantics to the labels used to annotate Information Objects and references.

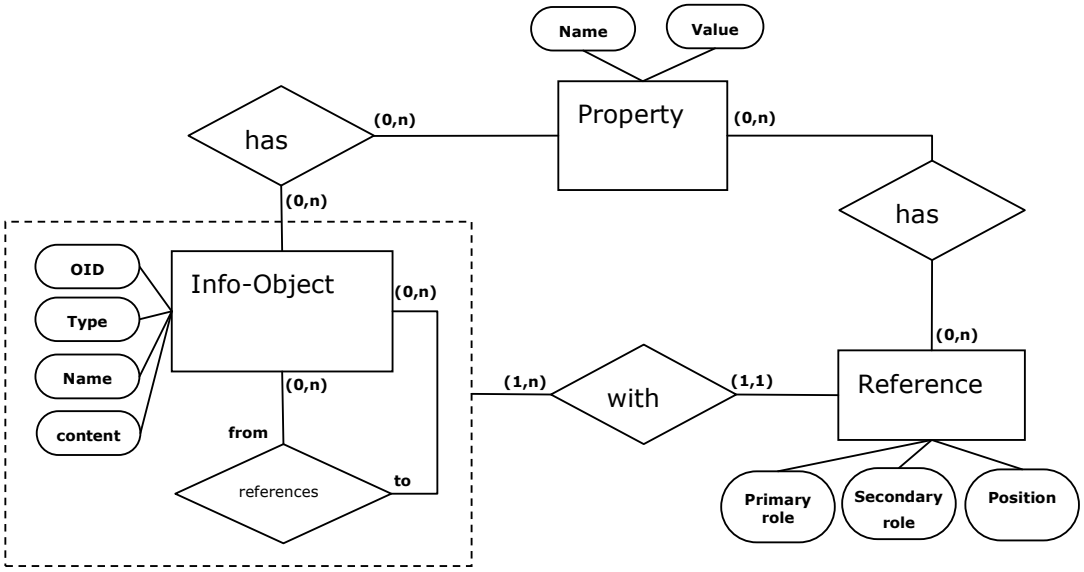

Figure 20. Information Object ER Model

6.1.2 Document and Collection Models

It is easy to build, starting from this model, a **document model** in which complex documents, composed of various, eventually nested subparts, are represented as chains of Information Objects linked via appropriate relationships. For instance, an HTML document that includes a number of images may be modelled as a complex object that provides references to Information Objects (containing the images). The positioning attribute present in the information-object model helps in representing an aggregate object made up of parts that have to be fitted together in a certain order. A dedicated component in the Information Organization family, the Content Management Service (cf. Section 6.4), exposes the document model to other services.

In a similar way, specific, complex metadata (like indexes, multimedia features) can be represented as separate Information Objects that are associated to the object they describe via appropriate relationships. For instance, a reference type may be "indexes" with a role name that gives additional information, like "full-text index".

The same representation mechanisms are also used to instantiate a concept of **collection.** Collections are the basic data structure used to organize information inside the Information Organization Services. Each collection is characterized by a collection identifier, labelled with a number of specific properties, and contains a number of documents. More specifically, a document can only exist as part of a given collection. Collections can in turn be nested, i.e. a collection can appear as member of another

that the target carries the same information as the source object, but presents it in a different form, e.g. using a different file format.

collection. A collection can be static (or *materialized*), that is contain a statically defined number of objects, that are added to it or deleted from it explicitly, or be *virtual.* The content of a virtual collection is not determined statically, but rather specified through declarative *membership predicates* that define which objects currently present in the gCube information-objects space are part of the collection. Its contents are thus determined dynamically at the moment when the collection is accessed by evaluating the membership predicates. For example, it is possible to define the collection of all objects having a certain MIME type (e.g. pdf).

6.1.3 Metadata and Annotation Models

The metadata and annotation models are based upon a number of characterisations of the primitives which define the gCube storage model, namely Information Objects and directed, binary relationships between such objects. Based on such characterisations, we can define the metadata and annotation models so as to follow closely the intuition and yet preserve a degree of semi-formal rigour.

- The *primary role* of a relationships is a characterisation of its intended semantics. We assume that the *secondary role* of a relationship is an *optional* specialisation of the primary role. Whenever convenient, we say that a relationship with (primary or secondary) role R is an *R-relationships* and that its source and target are an *R-source* or an *R-target*. If R1 and R2 are, respectively, the primary and secondary role of a relationship, then a R2-relationship is also an R1-relationship.

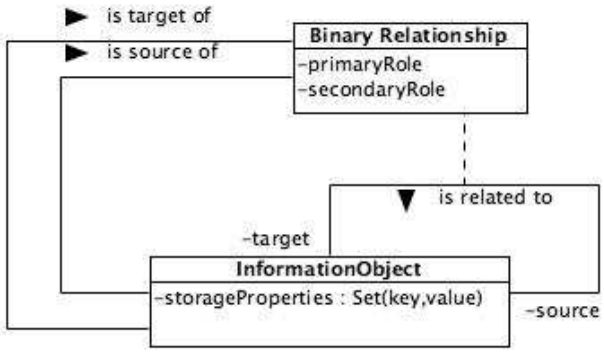

Figure 21. Relationship Model

- A relationships R is a *dependency* for its source (target) if the existence of the latter depends on the existence of at least one R-target (source). In this case, we also speak of a *dependent source* (*dependent target*). As a pragmatic corollary, an R-source (R-target) is deleted when its last R-target (R-source) is deleted.
- We say that a relationship is *exclusive* for its source (target) if it cannot relate the latter to more than one target (source). Otherwise, we say that it is *repeatable* on its source (target).

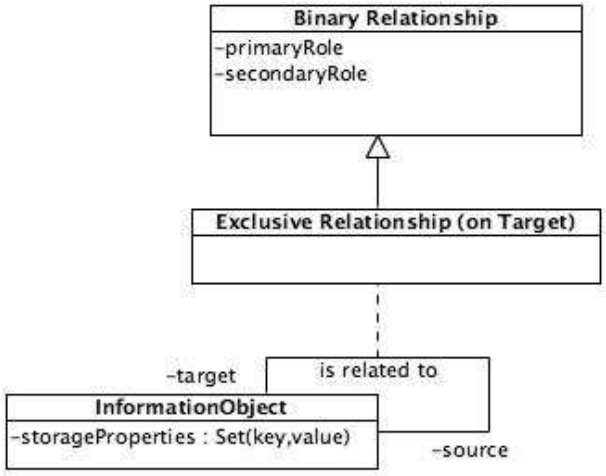

Figure 22. Exclusive Relations Model

- Relationships with primary role *is-member-of* (IMO) give the semantics of *collections* to their targets and that of collection *members* to their sources.
- We assume that IMO-relationships are repeatable on members, so that a object can be a member of more than one collection. We also assume that IMO-relationships dependencies on their members, so that a member is deleted when its last collection is deleted. Finally, we assume that collections cannot be members of other collections in turn.
- We expect the members of a collection to share enough similarities to be homogeneously processed, such as content formats and/or relationships. In particular, we speak of an *R-collection* to characterise a collection whose members are all sources (targets) of R-relationships.

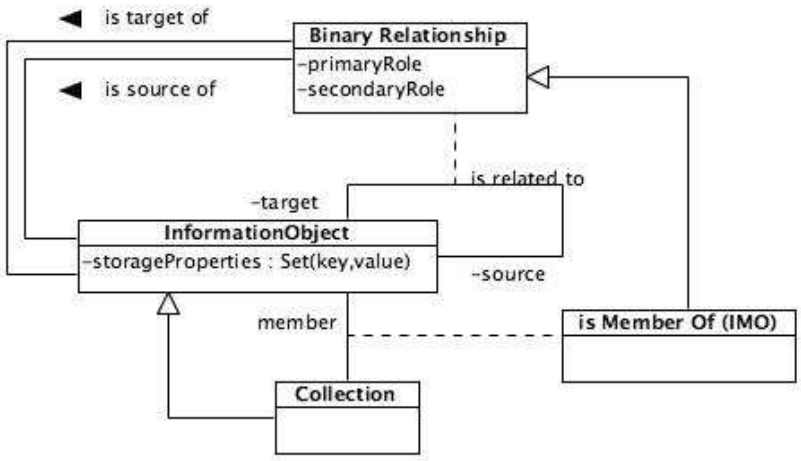

Figure 23. Membership Model

- We say that a relationship R *preserves membership* if it relates members of some collection C to members of the same or a different collection C'. If C is an R-collection, then we say that R is a *R-mapping* from C to C'.

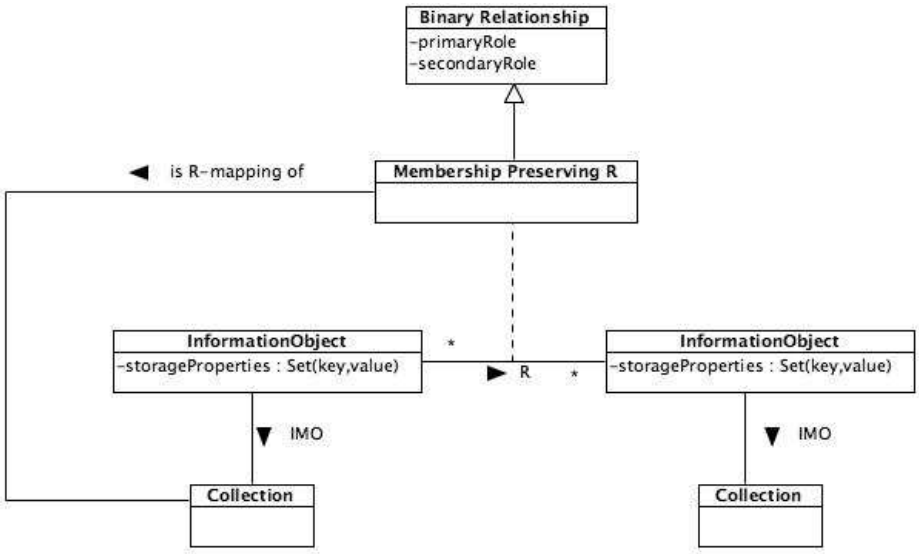

Figure 24. Membership Preservation Model

- Since objects can be members of multiple collections, we assume that relationships on members may hold in the *scope* of some but not all of those collections. Scope is most usefully and tractably modelled when it is lifted to entire collections. In particular, we say that a collection C is *in the scope* of a collection C' if there is an R-mapping from C to C' and there is an R-relationship between C and C'.
- We say that an object is a *document* if it models intellectual content. We then say that a collection is a document collection if all its members are documents.

6.1.3.1 The Metadata Model

- We identify a type of relationships with primary role *is-Described-by* (IDB) which give to targets the semantics of metadata about the corresponding sources. In particular, we say that an IDB-source is a *metadata object*, or simply metadata, for the corresponding IDB-target. We expect IDB-targets to be documents, but do not require it.
- We constrain IDB-relationships to be exclusive for their targets but repeatable for their sources. Accordingly, a metadata object can describe one and only one object but an object can have an arbitrary number of metadata objects. We also constrain IDB-relationships to preserve membership, so that a metadata object describes a member of some collection if and only if it is a member of some other collection in turn.
- We say that a collection M is a *metadata collection of type* R for a collection C if M is an R-collection in the scope of C for some secondary role R of IDB. Specifically, (i) all the members of M are metadata for members of C, and (ii) M is metadata for C.

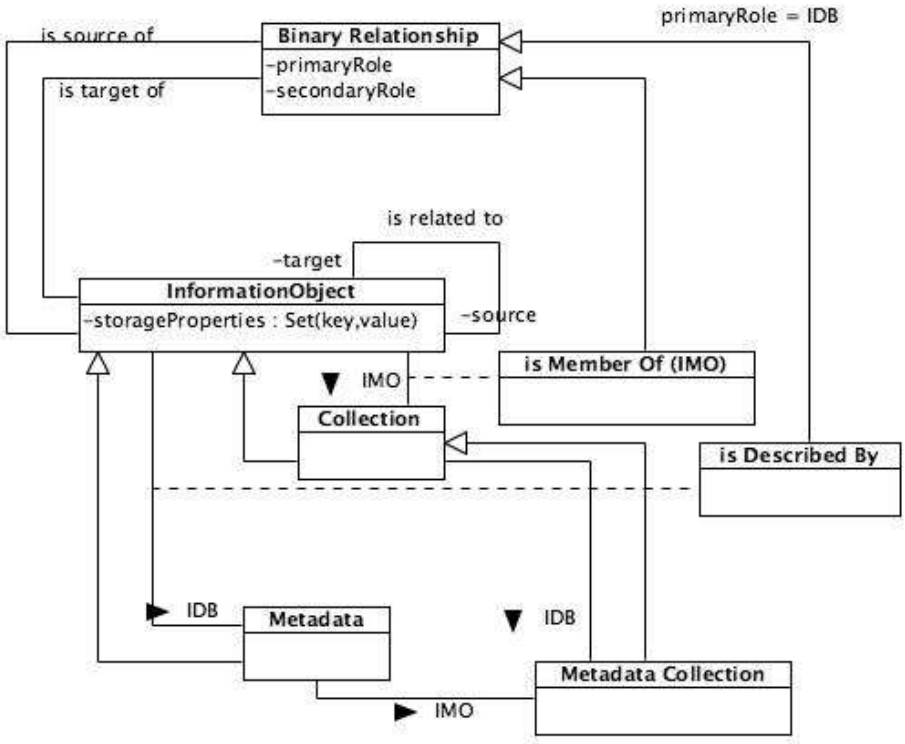

Figure 25. The Metadata Model

6.1.3.2 The Annotation Model

- We identify a secondary role *is-Annotated-by* (IDB) for the IDB relationship which give to targets the semantics of annotations of the corresponding sources. In particular, we say that an IAB-source is an *annotation object*, or simply annotation, for the corresponding IAB-target. We expect IDB-targets to be documents, but do not require it.
- We say that a collection M is an *annotation collection* if M is a metadata collection of type IAB.

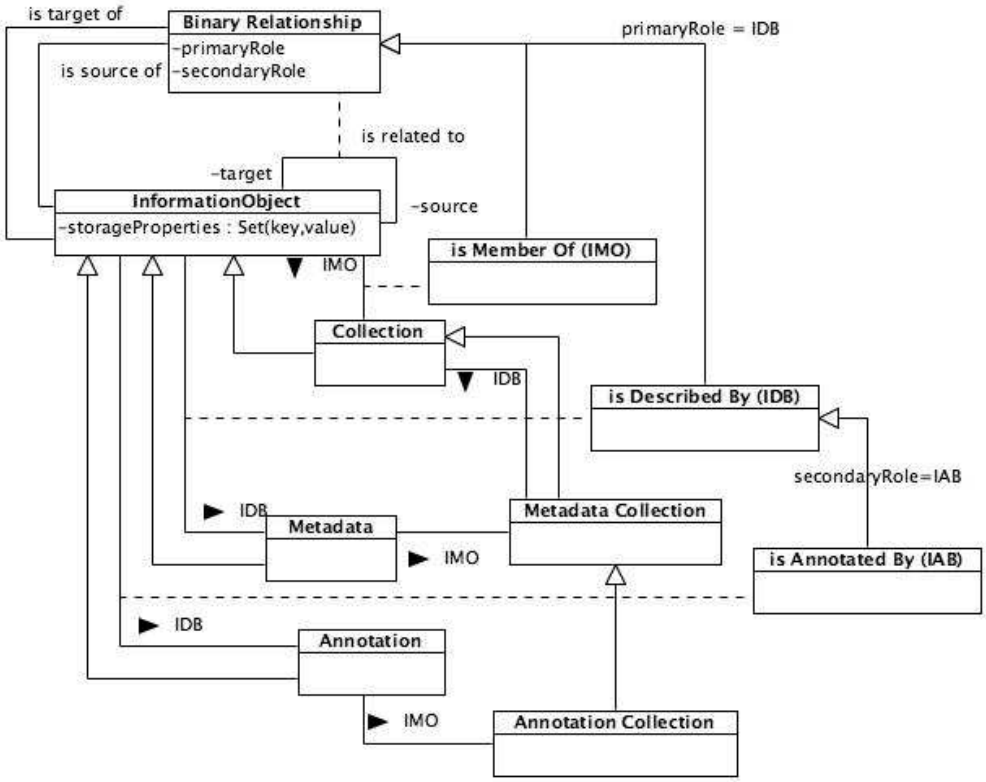

Figure 26. The Annotation Model

6.2 Overall Architecture

The Architecture of the Information Organization services is articulated over three fundamental layers, as illustrated in Figure 27, Base Layer, Storage Management Layer, Content Management Layer, Metadata and Annotation Management. Additional information about these layers and their functionality is provided below, while the technical details related to how to interact with the services at each layer are provided in the next sections.

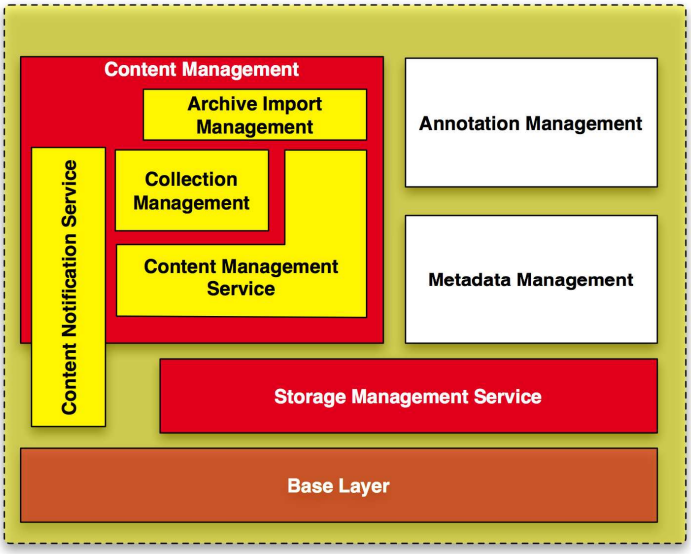

Figure 27. Data Management in gCube

The **Base Layer** is an abstraction from basic Grid storage facilities, as found in existing Grid infrastructures. Concretely, the Base Layer is a storage container responsible for storing the information needed to maintain the info-object model in a variety of storage facilities, like a hierarchical file system, a relational database and/or Grid storage facilities based on gLite, and to provide a uniform access to them to the layer directly above it. Though distinct from the upper layer from a logical and architectural point of view, the base layer exposes functionality only to the Storage Management Layer. For this reason, it functionality is only available as a Java API, used internally and not directly accessible through a service interface[9].

The **Storage Management Layer** is wrapped around the Base Layer and has the responsibility to (*i*) introduce Information Objects as an abstraction for manifold contentand associate storage properties to them; (*ii*) introduce generalized relationships between Information Objects, and managing their properties; (*iii*) preserve bindings to file-based content; (*iv*) maintain consistency of object relationships through appropriate constraints; and (*v*) provide a service interface for querying and updating the content, its associated storage properties and relationship information. This interface is exposed by a single service, the Storage Management Service.

The **Content**, **Metadata** and **Annotation Management** area contains the services the higher-level gCube services interact with for content, metadata and annotation manipulation. They are composed of a number of services that, building on top of the basic info-object model exposed by the storage management layer, provide high-level information-organization models/views. It is at this level that the actual semantics of entities used by gCube services (like collections, documents and metadata) are attached to the Information Objects and their relationships. In particular, the services that are part of this layer provide:

[9] From a software design point of view, to improve code modularity and maintainability, much of the functionality that is supplied by the Base, Storage and Content layers has been encapsulated in a set of libraries, organized in a layered fashion, closely following the logical architecture described above. Such libraries, going under the collective name of Content Management Library, are meant to make available functionality to Information Organization Services that use it internally and expose part of it through WSRF service interfaces, according to the SOA paradigm underlying gCube. As the Content Management Library is not thought for external use, but only as a support to the Information Organization Services, it is not described in detail here. Additional information on its features is provided in the documentation which is distributed with it.

- A document-oriented view of content, based on the document model described above. The functionality offered to manipulate documents includes basic document storage, lookup and update.
- Collection management functionality, as sketched above, used to organize documents into complex data sets.
- Fundamental metadata association.

Regarding notification facilities, services at higher level in the gCube infrastructure must be enabled to monitor changes at various levels in the content management layer. In particular, documents (as managed by the Content Management Service) and Collections (as managed by the Collection Management Service) should be monitored.

The services in charge of maintaining content resources must then propagate appropriate events corresponding to changes. Event Notification in gCube is based on WS-Notifications. Furthermore, the gCube infrastructure provides facilities for the brokerage of registrations to notifications. However, in the case of content management services it does not make sense to employ such facilities. As they manipulate an extremely large amount of objects it is not possible to publish individually each of these objects as a topic in the Information System. Thus, the services must expose directly methods that allow other services to register to specific kind of events on specific objects.

At this level is also present another service, that provides means to manage efficiently the functionality exposed by this layer. This takes care of *Archive import*, i.e. the creation and update of collections of documents starting from content available outside the infrastructure.

6.3 Base and Storage Management Layers

6.3.1 Reference Architecture

The Base and Storage Management Layers are exposed through a single service, the Storage Management Service. Internally, this service relies on the Content Management Library. Figure 28 shows the architecture of the Base and Storage Layers.

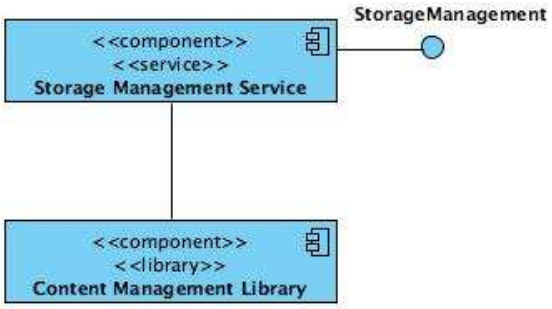

Figure 28. gCube Base and Storage Layers Architecture

6.3.2 Storage Management Service

The Storage Management Service encapsulates the functionality of the Base layer (which must not be exposed if not to the Storage layer) and that of the Storage Layer, and exposes it to other services in the Information Organization Family

This service provides the basic operations, manipulating, fetching, and deleting Information Objects according to the information-object model introduced above. This includes assignment of storage properties and set up of inter-object relationships.

The information internally maintained by the Storage Management Service amounts to all information required to describe existing Information Objects. The state of each Information Object is made up by its attributes and relationships (to other objects). This

information is partially stored in a relational DBMS. The logical schema of the database is depicted in Figure 29.

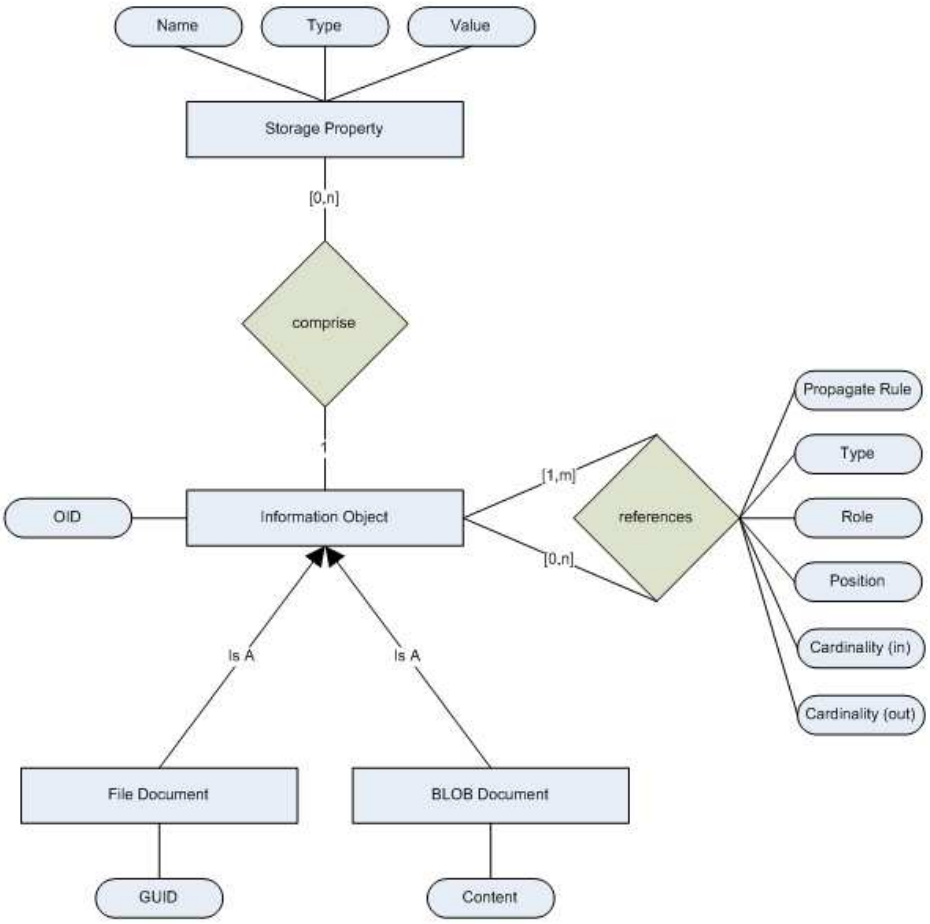

Figure 29. ER Schema Used to Instantiate the Information Object Model

This schema models the essential features of the information-object model introduced in the previous section. An Information Object is characterized by an OID, comprises a number of storage properties, can be seen as an abstraction over different types of contents and can reference other Information Objects via relationships.

It can be noticed that the *type* and the *name* of the Information Object are not modelled explicitly in this schema. This attributes are managed as if they were just another storage properties, and constrained (when necessary) only at the application level.

The other properties attached to an Information Object (and their intended semantics) are in general not defined in advance. However, the storage management itself attaches semantics to a number of properties which are used to handle or optimize the storage of info.objects and the access to them. These are:

- **Owner** identifies a gCube 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.
- **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 gCube storage but resides in the archive.

To relationships is also possible to attach general-purpose properties, in the form of type and role. The Storage Management Service also supports attaching to a relationship a delete-propagation rule (consulted upon removal of the referring object to determine whether to automatically delete the referred object). This facility provides efficient support for integrity constraints for for models build above the info-object model, that use references to represent complex information.

Several *propagation rules* have been defined. As mentioned above, all references are directed. Hence it is necessary not only to specify that deletion cascades, but also the direction in which this deletion is cascaded. The following rules are defined:

- *delete-target-propagate* – indicates that whenever the source object is deleted, the referenced object (the target object of the reference and the reference itself) will also be deleted and deletion cascades. Similar behaviour is also triggered, if not the source object, but just the reference is removed. It should be used carefully;
- *delete-target-if-single-appearance* – indicates that whenever the source object is deleted and the target object does not appear in the same role in another reference, then this target object is deleted, thus minimizing the danger of accidental object deletions;
- *delete-source-propagate* – indicates that whenever the target object is deleted, the source object of the reference (and the reference itself) will also be deleted and deletion cascades. Similar behaviour is also triggered, if not the source object, but just the reference is removed. Should be used with extreme caution;
- *delete-source-if-single-appearance* – indicates that whenever the target object is deleted, the source object will only be deleted, if it does not appear in the same role in another reference. For instance, if a document is member of several collections, it will not be deleted until the last collection which it references has been deleted.
- *no-delete-propagate* – indicates that no additional deletion is triggered, whenever this reference is deleted.

It is important to notice, however, that the storage manager is not responsible for maintaining the consistency of references among objects: the service ignores the semantics of roles introduced by higher-level services. This is the responsibility of the services that manage specific kind of objects (like complex documents). The Storage Management Service considers roles only in relation with the propagation of deletions.

Whenever data needs to be transferred, e.g. to download a document, it is necessary to define how the raw file content should be made available. For this, It is important to notice, however, that the storage manager is not responsible for maintaining the consistency of references among objects: the service ignores the semantics of roles introduced by higher-level services. This is the responsibility of the services that manage specific kind of objects (like complex documents). The Storage Management Service considers roles only in relation with the propagation of deletions.

Whenever data needs to be transferred, e.g. to download a document, it is necessary to define how the raw file content should be made available. For this, ***several transfer protocols are supported***. The preferred protocol is GridFTP (location starting with gsiftp://). Download of files is also supported for FTP (ftp://) and HTTP (http://) sites. Small files can be sent as part of a SOAP message (inmessage://) in base64 encoded form in the optional rawContent field. This transfer is limited by the GT4 container, which does support SOAP messages only up to a limited size of approximately 2 to 8 megabytes. Sending content as SOAP attachments is currently not possible, because GT4 will provide Attachment support starting with version 4.2. If the actual raw file content is not needed, e.g. because only the name of a document would be needed, a special location "/dev/null" can be specified. In this case, Storage Management will not perform any transfer at all, thus reducing the costs of this operation significantly.

In order to provide a flexible, yet stable API, optional parameters that do not affect directly the result of a service operation but only the way it is internally executed -(like specifying the best strategy to execute content transfer where more than one option is supported/available), can be supplied to many of the operations of the service in the

form of ***StorageHints.*** Storage hints are simply key-value-pairs. The names of these hints and their allowed values are specified in the API of the service. The rationale behind hints is to allow future extensions and modifications to the service while still maintaining stable its interface. Hints which are unknown to the service are simply ignored. On the other hand, omitting to specify a hint is never harmful, as the service will adopt a default behavios if not instructed otherwise. When an operation accepts hints, then it also provides a way to retrieve a list of *consumed hints,* specifiying which hints were actually used internally.

6.3.2.1 Resources and Properties

The Storage Management Service is implemented as a WSRF compliant stateless Web Service. While this service clearly maintains an internal state, this state is not related to the interaction within the service and a specific client, but coincides with the entire information space of the gCube infrastructure (i.e. all Information Objects stored in it). If the state of the service changes, the changes should be visible to all clients interacting with it. Furthermore, the mechanisms for maintaining state provided by the WSRF framework are not appropriate to handle in an efficient way the amount and complexity of the information the service has to maintain. For the same reason, the service does not publish any resource on the IS, as publishing a resource for each Information Object would yield a dramatic overhead to the system.

6.3.2.2 Functions

The main functions supported by the Storage Management Service are:

- **createInfoObject()** – which takes as input parameter a message containing the name and the type of the new Information Object to be created and returns the Information Object ID assigned to it;
- **removeInfoObject()** – which takes as input parameter a message containing the Information Object ID and removes it and its related relations from the Information Objects space managed by this service (that is, the information space of the whole gCube infrastructure);
- **removeInfoObjects()** – which takes as input parameter a message containing the list of Information Object IDs to be removed and returns the list of Information Object IDs representing the object that have not been removed and the error occurred;
- **hasRawContent()** – which takes as input parameter a message containing the Information Object ID and returns a Boolean value indicating whether the Information Object contains raw content associated to it or not;
- **getInfoObject()** – which takes as input parameter a message containing the Information Object ID, the URI representing a location the object will be stored and some storage hints and returns the relative Information Object;
- **getInfoObjects()** – which takes as input parameter a message containing a list of Information Object IDs, the URI representing a location each object will be stored and some storage hints and returns the relative Information Objects;
- **associateRawDocument()** – which takes as input parameter a message containing the Information Object ID, the URI the raw content can be downloaded or the raw content itself, and some storage hints and returns the storage hints consumed during the storage of the raw content;
- **updateRawDocument()** – which takes as input parameter a message containing the Information Object ID, the URI the raw content can be downloaded or the raw content itself, and some storage hints and returns the storage hints consumed during the update of the raw content;
- **removeRawDocument()** – which takes as input parameter a message containing the Information Object ID and removes the raw content associated to it;
- **addReference()** – which takes as input parameter a message containing a specification of the reference in terms of the reference *source* and *target* Information

Object (their IDs), the *role* of the referencing Information Object, the *secondary role* of the referencing Information Object (optional), the position this reference occupies in the referencing Information Object (optional) and the criteria governing the removal and creates this new reference between existing Information Objects;

- **addReferences()** – which takes as input parameter a message containing a list of reference specifications in terms of the reference *source* and *target* Information Object (their IDs), the *role* of the referencing Information Object, the *secondary role* of the referencing Information Object (optional), the position this reference occupies in the referencing Information Object (optional) and the criteria governing the removal; it returns a report on the result of the references requested by producing a list equal to the input parameter in which entries are enriched with a Boolean value indicating whether the operation has been successfully or not, the error message (if any) and the sequence number of the entry in the list;
- **removeReference()** – which takes as input parameter a message containing a specification of the reference in terms of the reference *source* and *target* Information Object (their IDs), the *role* of the referencing Information Object and, optionally, the *secondary role* of the referencing Information Object and removes the so identified reference from the existing references;
- **removeReferences()** – which takes as input parameter a message containing a list of reference specifications in terms of the reference *source* and *target* Information Object (their IDs), the *role* of the referencing Information Object and, optionally, the *secondary role* of the referencing Information Object and returns a report on the result of the reference removals requested by producing a list equal to the input parameter in which entries are enriched with a Boolean value indicating whether the operation has been successfully or not, the error message (if any) and the sequence number of the entry in the list;
- **retrieveReferences()** – which takes as input parameter a message containing the source Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference and returns the list of references matching these characteristics by specifying the source and target Information Object IDs, the role, the secondary role and the criteria governing the removal;
- **retrieveReferencesBulk()** – which takes as input parameter a message containing a list of reference specifications expressed in terms of the source Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference and returns the list of references matching these characteristics. As this is a bulk operation, it might fail only for some of the elements requested. For this reason, each entry in the result list specifies, besides the source and target Information Object IDs, the role, the secondary role, the criteria governing the operation and the sequence number of the entry in the list, also a Boolean flag denoting whether the operation was successful for the corresponding element and a field reporting an eventual related error message. For each entry, a client should first check the success flag and handle the entry according to its value;
- **retrieveReferenceTargetOIDs()** – which takes as input parameter a message containing a reference specification expressed in terms of the source Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference and returns the list of Information Object IDs of those objects referenced by the specified reference;
- **retrieveReferenceTargetOIDsBulk()** – which takes as input parameter a message containing a list reference specifications expressed in terms of the source Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference and returns the list of reference specifications expressed in terms of source Information Object ID, *role* of the reference, *secondary role* of the reference, target Information Object ID or error message, and sequence number of the entry in the list of those objects matching the specified criteria;
- **retrieveReferredAll()** – which takes as input parameter a message containing a reference specifications expressed in terms of the target Information Object ID, the

role of the reference and, optionally, the *secondary role* of the reference and returns the list of all the references currently existing that have the specified object as target specified in terms of the source and target Information Object IDs, the role, the secondary role, the criteria governing the removal;

- **retrieveReferredAllBulk()** – which takes as input parameter a message containing a reference specifications expressed in terms of the target Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference (these last two parameters are used to apply a filtering to the result) and returns the list of all the references currently existing that have the specified object as target; As this is a bulk operation, it might fail only for some of the elements requested. For this reason, each entry in the result list specifies, besides the source and target Information Object IDs, the role, the secondary role, the criteria governing the removal and the sequence number of the entry in the list, also a Boolean flag denoting whether the operation was successful for the corresponding element and a field reporting an eventual related error message. For each entry, a client should first check the success flag and handle the entry according to its value;
- **retrieveReferredSourceOIDs()** – which takes as input parameter a message containing a reference specifications expressed in terms of the target Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference (these last two parameters are used to apply a filtering to the result) and returns the list of all the Information Object IDs that are in a reference with the specified object as target;
- **retrieveReferredSourceOIDsBulk()** – which takes as input parameter a message containing a list of reference specifications expressed in terms of the target Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference (these last two parameters are used to apply a filtering to the result) and returns the list of references having the specified objects as target; this list of references is specified in terms of the target Information Object ID, the role, the secondary role, the source Information Object ID or an error message, and the sequence number of the entries in the list;
- **retrieveOIDInRelationWithAll()** – which takes as input parameter a message containing a reference specification consisting of a list of a Boolean value indicating whether the searched object is source or target in the reference, the Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference and returns a list on Information Object IDs of those objects that have a reference with all the objects in the specification;
- **retrieveObjectInRelationWithAll()** – which takes as input parameter a message containing a reference specification consisting of a list of a Boolean value indicating whether the searched object is source or target in the reference, the Information Object ID, the *role* of the reference and, optionally, the *secondary role* of the reference and returns a list on Information Object descriptions excluding their raw content;
- **setStorageProperty()** – which takes as input parameter a message containing the Information Object ID, the property name, the property type and the property value and sets the given property on the specified Information Object;
- **unsetStorageProperty()** – which takes as input parameter a message containing the Information Object ID and the property name and removes the given property of the specified Information Object;
- **retrieveStorageProperties()** – which takes as input parameter a message containing the Information Object ID and returns the properties attached to the given Information Object;
- **retrieveStorageProperty()** – which takes as input parameter a message containing the Information Object ID and the property name and returns the property name, value and type of the specified property;

- **retrieveOIDsByStorageProperty()** – which takes as input parameter a message containing the property name and the property value and returns the list of Information Object IDs of those objects having the specified property attached;
- **retrieveObjectsByStorageProperty()** – which takes as input parameter a message containing the property name and the property value and returns the list of Information Object Descriptions of those objects having the specified property attached;
- **retrieveOIDsHavingAllStorageProperties()** – which takes as input parameter a message containing a list of property names and property values and returns the list of Information Object IDs of those objects having all the specified properties attached;
- **retrieveObjectsHavingAllStorageProperties()** – which takes as input parameter a message containing a list of property names and property values and returns the list of Information Object Descriptions of those objects having all the specified properties attached;
- **createInfoObjectWithContent()** – which takes as input parameter a message containing the name and the type of the new Information Object to be created, the URI the raw content can be downloaded from or the raw content itself, some storage hints, a list of references with other Information Objects, and a list of properties and returns the Information Object ID of the just created object together with the consumed storage hints and a Boolean flag indicating whether the operation is successful or not;
- **createInfoObjectsWithContent()** – which takes as input parameter a message containing a list of Information Object specifications expressed in terms of the name and the type of each Information Object to be created, the URI the raw content can be downloaded from or the raw content itself, some storage hints, a list of references with other Information Objects, and a list of properties and returns the list of Information Object IDs of the just created object together with the consumed storage hints;

6.4 Content Management

6.4.1 Reference Architecture

The Content Management Layer exposes the functionality to manipulate documents and collections according to the corresponding models described above. It also offers facilities to populate collections starting from data which is stored externally to the gCube infrastructure. This functionality is provided by three main services *(i)* the Content Management Service; *(ii)* the Collection Management Service; and *(iii)* the Archive Import Service. Internally, these services use the Content Management Library. The architecture of this layer is shown in Figure 30.

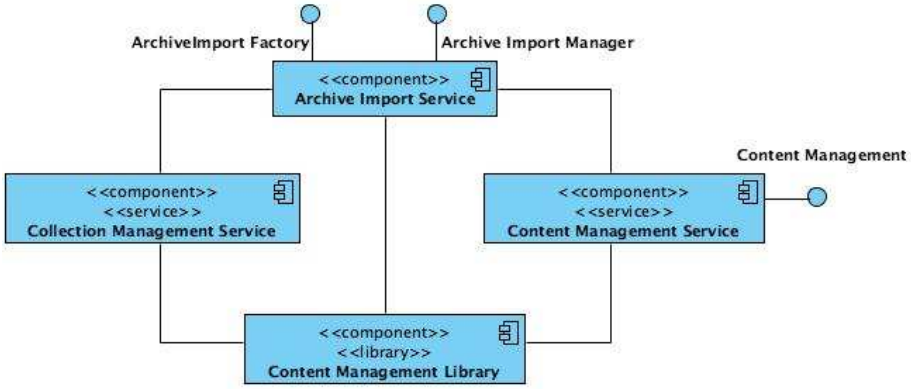

Figure 30. Content Management Services Reference Architecture

6.4.2 Content Management Service

This service expose the document model described in the previous section (cf. Section 6.1.2), providing operations for documents storage, access and update.

Internally, these high-level operations are mapped onto generic Storage Management operations. For this reason, many of its operations accept non functional parameters to be passed to the underlying Storage Management Service in the form of Storage Hints (see previous section). It is important to observe that, though the document model is per se independent of other high level models, this service is not completely independent, for the logics of its operation, from the collection Management Service described in the next section. More specifically, its interface enforces that any document must belong to at least one materialized collection. The reason for this is that permissions and visibility of resources are based on collections. Hence, a document cannot be stored outside of collections, because then its permissions would be undefined. Any document may on the other hand belong to more than one collection.

6.4.2.1 Resources and Properties

The Content Management Service is implemented as a stateless WRSF-compliant web service. It does not publish any resource, and only depends on the underlying Storage Management Service for its operation.

6.4.2.2 Functions

The main functions supported by the Content Management Service are:

- **storeDocument()** – which takes as input parameter a message containing the document ID, an URI from where the raw content can be gathered or the raw content itself, a collection ID and an array of storage hints and stores the document within the given collection;
- **getDocument()** – which takes as input parameter a message containing the document ID, the URI from where the raw content can be downloaded, a set of storage hints and a specification of the *unroll level*, i.e. an upper limit on the maximum number of relations that have to be traversed when retrieving a complex document composed by multiple nested subparts (this is an optimization feature), and returns the document description;
- **updateDocumentContent()** – which takes as input parameter a message containing the document ID, an URI from where the raw content can be gathered or the raw content itself, and an array of storage hints and updates the content of the given document;

- **deleteDocument()** – which takes as input parameter a message containing the document ID and deletes such a document (including raw content and references, may cause cascaded deletes based on propagation rules) from the set of documents managed by the service;
- **renameDocument()** – which takes as input parameter a message containing the document ID and the document name and attaches such a name to the given document;
- **setDocumentProperty()** – which takes as input parameter a message containing a document ID, a property name, a property type and a property value and attaches such a property to the given document;
- **getDocumentProperty()** – which takes as input parameter a message containing a document ID and a property name and returns the specified document property in terms of the ID, name, type and value;
- **unsetDocumentProperty()** – which takes as input parameter a message containing a document ID and a property name and removes the specified property from the document;
- **addPart()** – which takes as input parameter a message containing the parent document ID, the part document ID, the position the part should occupy in the parent document (optional) and a Boolean value indicating whether the associated part has to be removed whenever the main document is removed or not and attach the part to the specified document (i.e. creates the part-of relation);
- **removePart()** – which takes as input parameter a message containing the parent document ID and the part document ID and removes the specified part from the selected document (i.e. removes the part-of relation);
- **removeAlternativeRepresentation()** – which takes as input parameter a message containing the document ID and the representation ID, and removes the alternative representation from the selected document (i.e. the is-representation-of relation);
- **getParts()** – which takes as input parameter a message containing the parent document ID and returns the list of Document IDs that are (direct) parts of the specified document;
- **getDirectParents()** – which takes as input parameter a message containing an Information Object ID and returns a list of Information Object IDs of all the objects that are (direct) parents of the specified one;
- **getDirectParents()** – which takes as input parameter a message containing the ID of an alternative representation and returns the list of Information Object IDs (OIDs) of the main representations of the Document;
- **addAlternativeRepresentation()** – which takes as input parameter a message containing the document ID, the ID of an Information Object that represent the alternative representation, the rank this alternative representation has (optional) and a Boolean value (optional) indicating whether the alternative representation has to be deleted whenever the main object is deleted or not, and assigns such a representation to the specified Document;
- **getAlternativeRepresentations()** – which takes as input parameter a message containing the document ID and returns the list of all existing representations of the specified document;
- **hasRawContent()** – which takes as input parameter as input parameter a message containing the document ID and returns a Boolean value indicating whether the document has a raw content or not;
- **getContentLength()** – which takes as input parameter as input parameter a message containing the document ID and returns the size of the raw content of the document;
- **getMimeType()** – which takes as input parameter a message containing the document ID and returns the MIME Type associated to the raw content of the document;

- **setMimeType()** – which takes as input parameter a message containing the Document ID and a string representing a MIME Type and set such a type as those of the raw content of the document;
- **registerToDocumentEvent()** – which takes as input parameter a message containing the DocumentID, the event type and the EPR of a service and registers the service to that kind of event on the specified document.

6.4.3 Collection Management Service

This service provides the high-level operations for collection management which are mapped onto generic Storage Management operations.

6.4.3.1 Resources and Properties

The service is implemented as a stateless WRSF service. However, it publishes summary information about the collections it manages on the Information System. Each collection is exposed as a distinct gCube-Resource. The properties stored in the resource are the ID and name of the collection, its storage properties (e.g is the collection is virtual and if it is a user-collection), and its references.

6.4.3.2 Functions

The main functions supported by the Collection Management Service are:

- **createCollection()** – which takes as input parameter a message containing a collection name, a Boolean flag indicating whether the collection is virtual or not, and a membership predicate (optional) returns the Information Object ID of the object created to represent the collection;
- **getCollection()** – which takes as input parameter a message containing a collection ID and returns the collection description including the object ID, the object type, the object name, the object flavour, the virtual flag, the user collection flag, the membership predicate, the created and last modified value, and the set of other collection properties of the given collection;
- **getMembers()** – which takes as input parameter a message containing a collection ID and returns a list of Information Object Description (including the object ID, the object type, the object name, the object flavour, the virtual flag, the user collection flag, the membership predicate, the created and last modified value, and the set of other collection properties) of all the objects members of the given collection;
- **getMemberOIDs()** – which takes as input parameter a message containing a collection ID and returns a list of Information Object IDs (OIDs) of all the objects members of the given collection;
- **addMember()** – which takes as input parameter a message containing the collection ID and a Document ID and adds the object to the given collection. This method fails if invoked on a virtual collection;
- **removeMember()** – which takes as input parameter a message containing a collection ID and a Document ID and removes the specified object from the given collection. This method fails if invoked on a virtual collection;
- **updateCollection()** – which takes as input parameter a message containing the collection ID and a membership predicate and changes the collection accordingly. For a virtual collection, this simply updates the stored membership predicate. For a materialized collection, it adds all new members that satisfy the predicate and removes all members that no longer satisfy the predicate. The latter may cause deletion of documents that are no longer member of any collection;
- **deleteCollection()** – which takes as input parameter a message containing the collection ID and removes such a collection from the set of collections existing in the gCube system;
- **renameCollection()** – which takes as input parameter a message containing a collection ID and a collection name and assigns the specified name to the given collection;

- **setCollectionProperty()** – which takes as input parameter a message containing a collection ID, a property name, a property type and a property value and attach such a property to the given collection;
- **getCollectionProperty()** – which takes as input parameter a message containing a collection ID and a property name and returns the specified collection property, i.e. the object ID, the property name, type and value;
- **unsetCollectionProperty()** – which takes as input parameter a message containing a collection ID and a property name and removes the specified parameter from the ones attached to the given collection;
- **getMaterializedCollectionMembership()** – which takes as input parameter a message containing a document ID and returns a list of collection IDs and names of the materialized collection such an object is member of;
- **isUserCollection()** – which takes as input parameter a message containing the collection ID and returns a Boolean value indicating whether the given collection is a user collection, i.e. can be directly perceived by the VRE users, or not;
- **setUserCollection()** – which takes as input parameter a message containing a collection ID and a Boolean value indicating whether the specified collection has to be considered a user collection, i.e. a collection that is perceived by the users of some Virtual Research Environment, or not and sets such a property on the given collection;
- **listAllCollectionIDsAndNames()** – which returns a list of collection ID and collection name representing all the collections currently managed in gCube which are defined in the scope of the caller;
- **listAllCollections()** – which returns a list of collection descriptions representing all the collection currently managed in gCube which are defined in the scope of the caller;
- **listAllCollectionsHavingNames()** – which takes as input parameter a message containing a collection name and returns a list of collection descriptions of the collections having such a name. No support for wildcards of similar kind of similarity evaluation. This method and the following one are provided for those services who need to access collections by name instead that by their ID;
- **listAllCollectionIDsHavingName()** – which takes as input parameter a message containing a collection name and returns a list of collection IDs of the collection having such a name. No support for wildcards of similar kind of similarity evaluation;
- **registerToCollectionEvent()** – which takes as input parameter a message containing the CollectionID, the event type and the EPR of a service and registers the service to that kind of event on the specified collection;

6.4.4 Archive Import Service

The Archive Import Service (AIS) is in charge of defining collections and importing their content into the gCube infrastructure, by interacting with the Collection Management service, the Content Management service and other services at the content management layer, such as the Metadata Manager Service. While the functionality it offers is, from a logical point of view, well defined and rather confined, the AIS must be able to deal with a large number of different ways to define collections and offer extensibility features so to accommodate types of collections and ways to describe them not known/required at the time of its initial design. These needs impact on the architecture of the service and on its interface. From an architectural point of view, the AIS offers the possibility to add new functionality by using pluggable software modules. At the interface level, high flexibility is ensured by relying on a scripting language used to define import tasks, rather than on a interface with fixed parameters only.

As importing collections might be an expensive task, resource-wise, the AIS offers features that can be used to optimize import tasks. In particular, it supports incremental import of collections. The description that follows introduces first the rationale behind the functionality of the AIS, its overall architecture, its scripting language, its

extensibility features, and the concepts related to incremental import. Then, it presents the interface of the service.

Overall, the execution of an import task can be thought as divided in two main phases (*i*) the *representation phase* and (ii) the *import phase*.

During the ***representation phase***, a representation of the resources to be imported and their relationships is modelled using a graph-based model. The model contains three main constructs: *collection*, *object* and *relationship*, thus resembling the collection and Information Object models on which the content management services are based. Each construct has a type, and can be annotated with a number of properties. The type of the resource is a marker that is used by importers to select the resources they are dedicated. The properties are just name-value pairs. These values can be of any java type. Only some types of resources and their properties are fixed in advance. The existing types are used to support the import of content and metadata. The functionality of the AIS can be extended define new types, annotated with different properties. To specify a representation of the resources to import it is possible to use a procedural scripting language called *AISL* (*Archive Import Service Language*). More details on the language are given below.

The *representation graph* built during the representation phase is then exploited during the ***import phase***. An import-engine dynamically loads and invokes a number of *importers*, software modules that encapsulate the logics needed to import specific kind of resources, interfacing with the appropriate services. Each importer inspects the representation graph, identifies the resources for which it is responsible, and performs the actions needed to import them. Besides performing the import, the importer may also produce some further annotation of the resources in the graph. These annotations are used in later execution of tasks that involve the same resources, and may also be exploited by other importers. For example, in the case of importing metadata related to some content, the content objects to which the metadata objects refer should have already been stored in the content-management service, and should have been annotated with the identifiers used by that service. Similar considerations hold for content collections. Importers enabled to handle content-objects and metadata objects are already provided by the AIS. Additional importers, dedicated to store specific kind of objects, can be added. At the end of the import phase, the resource graph created during the representation phase and annotated during the import phase is stored persistently.

Defining an import task for the AIS amounts to building a description of the resources to import. This can be done by submitting to the AIS a script written in AISL. ***AISL*** is an interpreted language with XML-based syntax, designed to support the most common tasks needed during the definition of an import task. As most programming languages, it supports various flow-control structures, allows to manipulate variables and evaluate various kinds of expressions. However, the goal of an AISL script is not that of performing arbitrary computations, but to create a graph of resources. Representation objects (collections, objects and relationships) are first class entities in the language, that provides constructs to build them and assign them properties. Representation objects may resemble objects in oo-languages, in that their properties can be accessed as fields and assigned values, and references to representation-objects themselves can be stored in variables. A fundamental difference is that, once created, representation objects are never destroyed, even when the control-flow exits the scope in which they were created.

AISL is not typed, i.e. variables can be assigned values of any kind, and is the responsibility of the programmer to ensure proper type concordance. Basically, AISL does not even define its own type system, but exploits the type system of the underlying Java language. The language offers constructs for building objects of some basic types, like strings, sets, files, and integers. However, expressions in AISL can return any Java object. Specific functions can be thus used to build objects of given types in a controlled way. For instance, the built-in "dom()" function accepts as input a file object and produces a DOM document object. Similarly, the "xpath" function takes a

DOM object and an expression, and returns the result of the evaluation of an xpath over the DOM node object as a set of DOM Node objects. In general, thus, types cannot be directly manipulated from the language, except for a few cases, but the variables of the language can be assigned with any java type, and objects of given type can be built using functions. This means that add-on-functions can produce as result objects of user-defined types. These objects can be stored in representation objects as properties, and later used by specific add-on-importers. However, AISL itself provides a few special data-types, not present in the standard java libraries. In particular, the type AISLFile is designed to work in combination with some AISL built-in functions and to optimize the management of files during import, in particular with regard to memory and disk storage resources consumption. An AISLFile object encapsulates information about a given file, such as file length, and allow access to its content. However, the file content is not necessarily stored locally. If the file is obtained through the file() function, then the download of its actual content is deferred, and only performed when needed. When describing a large number of resources, as in the case of large collections, it is not feasible (or anyway not efficient) to store locally to the archive import service all contents that need to be imported. This is especially true for content that has to be imported without having to be processed in any way before the import itself. Even for those files that might need some processing (for instance for extracting information), it might be desirable for the AISL script-writer to be able to import the file, use for the time he needs it (e.g. pass it to the xslt() function) and then free the memory resources used to maintain it, without having to deal directly with the details. The AISL file offers a transparent mechanism to handle access to the content of remote files accessible through a variety of protocols, by having a placeholder (an AISFile object) that can be treated as a file. Internally, an AISLFile implements a caching strategy for documents whose content has to be accessed at resource-description time. For files whose contents have to be handled only at import time, it offers a way to encapsulate easily inside a representation object all the information needed to pass to other service like the storage management or the content management service.

To encapsulate complex operations, AISL provides functions that work as in most programming languages. A number of built-in functions is already defined inside the language. These functions cover common needs arising during the definition of resource graphs.,.For instance, the dom() function takes as input an AISFile object and returns a DOM document obtained by parsing the file (or null if the parsing fail, e.g. if the file is not a valid XML file). The language can be expanded by adding new functions. Furthermore, AISL provides *extensible-functions*, meaning that their functionality can be extended to handle special kinds of arguments. For instance, the file() function allows to retrieve a list of files given a set of arguments that define their location. The built-in function is already able to deal with a number of protocols, including http, ftp and gridftp. However, the function can be extended to handle additional protocols. The motivation behind extensible functions is to keep the syntax of AISL as lean and transparent for the user as possible.

The complete syntax of the AISL language and the reference on its built-in functions, together with examples of usage, are detailed in the documentation accompanying with the service.

The entire architecture of the AIS is design to offer an high degree of extensibility at various levels. All these mechanisms are based on a plug-in style approach: the service doesn't have to be recompiled, redeployed or even stopped whenever an extension is added to it.

An AISL script can be used to specify an AIS import task in a fully flexible way. However, it might be desirable to offer a simpler interface to certain functionality that has to be invoked often without many variations. For instance, in the case when the files to import are all stored at certain directory accessible through ftp, or in the case when a single file available at some location describes the location of files and related metadata, it would be desirable to invoke the AIS with the bare number of parameters needed to perform the task, rather than having to write an AISL script from scratch. For this

reason, it is possible to plug-in in the AIS *adapters*. An adapter is a software module that can be invoked directly from the interface of the AISL with a limited number of parameters and is in charge of building a representation graph. Internally, the adapter may use a parameterized AISL script to perform its job, but also manipulate directly the constructs of the representation model.

The AISL language itself provides extension mechanisms. New functions can be defined for the language, and existing extensible functions can be extended to treat a larger number of argument types. The representation model used to describe resources is fully flexible: new types can be defined for collections, objects and relationships, and all these representation constructs can be attached with arbitrary properties of arbitrary types.

Regarding the importing phase, new objects types can be handled by defining pluggable importers. These importers will be invoked together with all other importers available to the AIS service, and handle specific types of representation objects. It is to be observed that the definition of new representation constructs and that of new importers are not disjoint activities. In order for new object types to have meaning, there must be an appropriate importer which is able to handle them (otherwise they will just be ignored). In order for an importer to work with a specific kind of object type, the importer must be aware of its interface, i.e. its properties. While properties attached to representation objects can be always accessed by name, in order to make easier the development of importers it is possible to define new representation constructs so that their properties can be accessed via getters and setters. During the description phase, the AISL interpreter will recognize if a new type is connected to a subclass of the related representation construct and build an object of the most specific class, so that later on the importers can work directly on objects of the types they recognize.

In order to support these extensibility mechanism, the AIS exploits a simple *plug-in pattern*. The modules (Java classes) that have to be made available must extend appropriate interfaces/classes defined inside the AIS framework, and be defined in specific Java packages. To be compiled properly, these classes must of course be linked against the rest of the code of the AIS.

After compilation, the resulting .class files can be made available to an AIS instance by putting them into a special folder under the control of the AIS instance. The classes will be dynamically loaded and used (partially using the Java reflection facilities).

The archive import service supports *incremental import*. With this term, we denote the fact that if the same collection is imported twice, the import task should only perform the actions needed to update the collection with the changes occurred within the two imports, and not re-import the entire collection. Incremental import requires two features. First, it must be possible to specify that a collection is the same an another collection already imported.

For the existing importers, this is achieved by specifying inside the description of a collection the unique identifier of the collection in the related service. For instance, for content collections, this would be the collection ID attached to each collection by the Collection Management Service. After the description of the new collection has been created, the service will compare this description with that resulting from the previous import, and decide which objects must be imported, which must be substituted, and which must be deleted from the collection.

When comparing two collections, the import service must know how to decide whether two objects present inside a collection are actually the same object. In order to support this behaviour, the AIS must support two concepts, that of external-object-identity and that of content-equality.

External object identity is an external object-identifier. Two objects are considered to be the same if and only if they have the same external identifier. Notice that the external identifier is distinct from the internal-object identity that is used by services to distinguish between objects. Of course, the AIS must ensure a correspondence between internal and external identifiers. Thus, if an object with a given external identifier has been stored with a given internal identifier, then another object is imported with the

same external identifier then the AIS will not create a new object, but (possibly) update the contents/properties of the existing object.

External Identity is sufficient to decide whether an object to be imported already exists in a given collection. If an object already exists, and it has changed, then it will be updated. Deciding whether an object has changed also requires additional knowledge. For many properties attached to an object, the comparison is straightforward. However, for the actual content of an object this is less immediate. The content of an object (that is a file) might reside at the same location, but differ from that previously imported. Furthermore, comparing the old content with the new content is an expensive task: it requires at least to fetch the entire content of the new object and thus is as expensive as just re-importing the content itself. For this reason the AIS also supports the concept of content-identity.

A content-identifier can be specified. If two identical objects (i.e. having the same external-object-identity) have the same content-identifier, than the content is not re-imported. If, on the other hand, the identifier differs, then the new content is re-imported. A content identifier can be a signature of the actual content, or any other string. If a content identifier is not specified, the AIS the content-identifier is obtained exploiting the following information location of the content, size of the content (when available, depends on the particular protocol used), date of last modification of the content (when available, depends on the protocol used), an hash signature of the content (when available from the server, depends on the protocol used. For instance, it is possible to get an MD5 signature of a file from an HTTP server).

The AIS also supports *continuous import*. The term continuous import denotes the fact that the AIS should provide updates for certain collections without need to resubmit the same import task every time. As a way to monitor resources outside the gCube infrastructure is not defined, this means that the same import task will not be executed once, but scheduled for re-execution at appropriate intervals of time.

6.4.4.1 Resources and Properties

The Archive Import Service is implemented as a stateful WSRF-compliant web-service, following a factory-instance pattern, and publishes a WS-resource for each instance. The WS-resource contains the parameters that are used to create the instance (e.g. the related AISL script), but also contains summary information like the current status of an import (on-going, completed, continuous), the collections involved in the import, information on errors and other information useful for monitoring the import of collections.

6.4.4.2 Functions

The main functions supported by the Archive Import Service Factory are:

- **createImportTask()** – which takes as input parameter a message containing a specification of an import action expressed through the AISL scripting language, a specification whether the import should be continuous and an interval of time for the re-execution of the task, and submits it to the appropriate agent managing the import action, returns the EPR of the corresponding instance;
- **createAdapterTask()** – which takes as input parameter a message containing an adapter component (i.e. an helper implementing the import action accordingly) and the relative parameters, a specification whether the import should be continuous and an interval of time for the re-execution of the task, and submits this import action task to an appropriate agent, managing the import action through the given adapter, and returns the EPR of the corresponding instance;
- **registerPlugin()** – which takes as input parameter a message containing the URI a new import plug-in can be downloaded and loads such a plug-in to the pool of those equipping the service. The plug-ins will be available to all instances created by the factory afterwards.

The main functions supported buy the Archive Import Manager (i.e. the service acting on the WS-Resource) are:

- **stopImportTask()** – which takes as input parameter a message specifying a stop rule, and causes the import action performed by the instance to stop. This method is provided to control instances that run a continuous import task.

6.5 Metadata Management

In gCube, the Metadata Management has been designed to rely on the storage facilities provided by the Storage Management layer and its support to manage primary and secondary relations between Information Objects (cf. Section 6.1.3). The subsystem is called upon to support the management of metadata objects and metadata collections, i.e. supporting their creation, access, storage and removal of objects compliant with one or more metadata format and their efficient search. To implement such functions the internals of the subsystem has been organised according to the following architecture.

6.5.1 Reference Architecture

The functions expected by the Metadata Management subsystem are implemented by two cooperating services: one taking care of the creation and management of Metadata Objects and Metadata Collections (Metadata Manager) and another taking care of the support to metadata object retrieval (XML Indexer) as depicted in Figure 31.

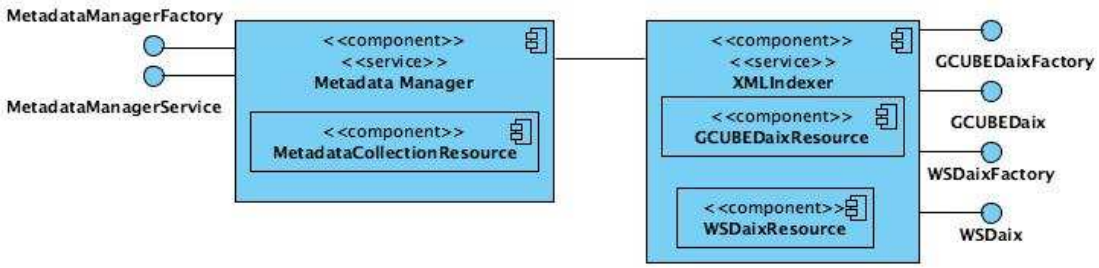

Figure 31. gCube Metadata Management Reference Architecture

- **MetadataManager** – this Service is responsible for managing the incoming requests for changing, accessing and manipulating metadata. It associates metadata with Information Objects, remove associations, returns metadata for a given object and updates associated metadata;
- **XMLIndexer** – this Service is a generic XML Indexer service allowing to query the metadata by resolving arbitrary XQuery and XPath expressions against its managed content. It indexes homogeneous collections of XML data.

6.5.2 MetadataManager

The Metadata Manager is a gCube Service that models arbitrary metadata relationships (IDB-relationships). The only assumption it does is that the metadata objects are serialized as well-formed XML documents. The service has a two-fold role:

- to manage Metadata Objects and Metadata Collections;
- to establish secondary role-typed links. Such relationships can be in place between any type of Information Object and in the scope of a Collection or not.

The main functionality of the Metadata Manager components is the management of Metadata Objects, Metadata Collection and their relationships. To operate on Metadata Collections, the Metadata Manager instantiates a Collection Manager for each collection. A Collection Manager is the access point to all the possible operations on a specific Metadata Collection. From an architectural point of view, the Metadata Manager adopts the Factory pattern and Collection Managers are implemented as a stateful WS-Resource. Physically, the service is composed by:

- the **MetadataManagerFactory**, a factory service that creates new Collection Managers and offers some cross-Collection operations
- the **MetadataMetadataManager**, a service that operates over Metadata Collections and on Metadata Objects as Elements, i.e. members of a specific Metadata Collection

6.5.2.1 Resources and Properties

The Service adopts a Factory Pattern and creates a WS-Resource for each Metadata Collection.

The Metadata Manager Factory creates new Collection Managers and offers some cross-Collection operations. Moreover, it operates on Metadata Objects as Information Objects related to other Information Objects and not as Members of Metadata Collections.

6.5.2.2 Functions

The main functions supported by the Metadata Manager Factory are:

- **createManager()** – which takes as input parameter a message containing a set of creation parameters and a Collection ID and creates a new Manager in order to manage a Metadata Collection bound to such a collection. If a Metadata Collection with the specified Metadata characteristics does not exist, the Manager creates the Metadata Collection, binds it with the Document Collection with the given secondary role relationship and publishes its profile in the IS.
- **createManagerFromCollection()** – which takes as input parameter a Collection ID and return the Collection Manager for that Collection if the Metadata Collection with the given ID exists. It returns an error if the Metadata Collection with the given ID does not exist.

The Metadata Manager Service operates on a Metadata Collection. This means that the following operations always executed over the Metadata Collection specified at Manager creation time and its members.

The main functions supported by the Metadata Manager Service are:

- **addElements()** – which takes an array of couples (an Information Object ID and a Metadata Object) and for each array element stores the Metadata Objects on the Storage Management Service as Information Object;
- **addElementsRS()** - which implements the previous method by reference;
- **deleteElements() -** which deletes from the current Metadata Collection the elements specified in the list specified as parameter;
- **getElements() -**which takes a list of Object ID and returns the set of associated Metadata Objects.

6.5.3 XMLIndexer

The XMLIndexer Service is a generic indexer of XML data homogeneous collections. The service allows creating, populating and resolving queries against such collections. Two types of XMLIndexer have been designed, each of them manages a collection of XML documents:

- ***WSDaix*** – a WSDaix is completely unconscious of the collection and the data handled. This means that it does not impose any constraint about them and, therefore, it assumes that the clients know the schema of the documents to query. It can be used each time it is useful to index and query a (temporary) set of XML data, like a result set;
- ***GCUBEDaix*** – a GCUBEDaix is bound to a specific Metadata Collection and it is used to index the elements of such a collection. When a new Metadata Collection to be indexed is created, the Metadata Manager Service creates also a new related GCUBEDaix and, each time a new Metadata Object is added/updated in such collection, the Metadata Manager Service also adds/updates the GCUBEDaix by feeding it with the new element.

6.5.3.1 Resources and Properties

The Service adopts a Factory pattern and creates a WS-Resource per each XML Indexer. Since there are two kinds of XML Indexer, there are also two kinds of WS-Resource that the Factory service can create: the GCUBEDaix resource and the WSDaix resource. The state of each Indexer is published in the DIS by means of its WS-ResourceProperties. These resource properties includes the creation parameters and, if the Indexer is a WSDaix, the SetTerminationTime and CurrentTime WS-ResourceProperties.

A GCUBEDaix operates on a collection of homogeneous XML documents bound to a specific Metadata Collection. Since the managed XML documents are wrapped in the Metadata envelope, each document is identified by a unique ID (the Metadata Object ID) and this allows a more advanced management of this type of Indexer with respect to the WSDaix one.

6.5.3.2 Functions

The main functions supported by the GCUBEDaix XMLIndexer are:

- **addElements()** – which takes a list of XML documents and adds them to the current XML collection managed by this instance;
- **addElementsRS()** – which implement the previous operation by reference, i.e. the message passed as input parameter contains the EPR of a result set containing the list of documents;
- **executeXPath()** – which takes and executes a valid XPath expression against the current XML collection and returns all the XML Documents that match the given expression;
- **executeXPathRS()** – which takes and executes a valid XPath expression against the current XML collection and returns a reference (the EPR of a Result Set containing them) to the list of XML Documents that match the given expression;
- **executeXQuery()** – which takes and executes a valid XQuery against the current XML collection and returns the list XML Documents resulting from the evaluation of the given query on the managed collection;
- **executeXQueryRS()** – which takes and executes a valid XQuery against the current XML collection and returns a reference to (its EPR) the list XML Documents resulting from the evaluation of the given query on the managed collection;

The main functions supported by the WSDaix are those specified in the WS-DAIX specification [3].

6.6 Annotation Management

In gCube, the management of annotations involves front-end and back-end components designed for, respectively, interactive and programmatic consumption. The two classes of components are strongly related, components that present annotations interact directly with those that instead help organising them. This section focuses exclusively on back-end components, while front-end components are discussed in Section 8.4.8.

6.6.1 Reference Architecture

Annotation management is the sole responsibility of the **Annotation Back-End** (ABE) service. Precisely, the ABE service manages the entire lifecycle of *annotation relationships* between Information Objects (cf. Section 6.6.1.1) – from their creation and collation to their retrieval, update, and deletion – independently from application-specific models of annotation content (cf. Section 6.1.3.2).

Architecturally, the service is placed at the top of a stack of Information Organisation services, as shown in Figure 27. The stack is built out of Storage Management services (cf. Section 6.3), Content Management services (cf. Section 6.4), and Metadata Management services (cf. Section 6.4.4). Its main role is one of mediation between the users of Information Presentation services and Metadata Management services one step down the stack. In this role, the value it offers to its clients is one of specialisation and, most noticeably, transparent aggregation of functionality available more generically for

the management of metadata. An overview of the design and functionality of the ABE service is given in Section 6.6.2.

The **Annotation Back-end Library** is a library that ABE clients *may* use to simplify their interaction with the service. It can be thought of as an extension of the client-side library that gCube services offer when distributing stubs generated automatically from public service interfaces. An overview of the design and functionality of the ABE library service is given in Section 6.6.3.

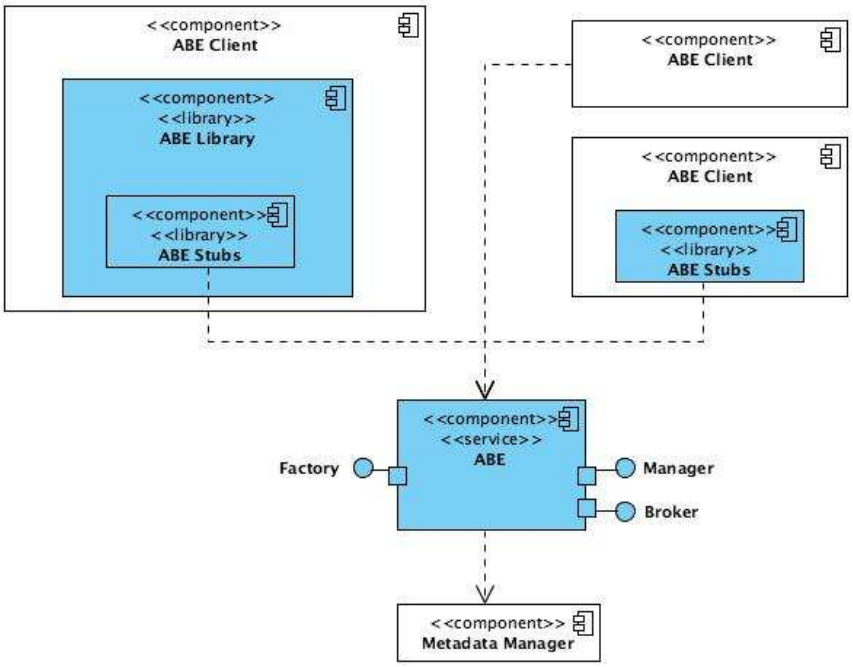

Figure 32. Annotation Back-end Service and Library

6.6.1.1 Annotation Relationships

Annotation relationships are specialisations of metadata relationships that give target objects the broad semantic of subjective and contextual assertions about source objects. More formally, annotation relationships are binary relationships with primary role *is-described-by* and secondary role *is-annotated-by* (IAB).

As specializations of metadata relationships, annotation relationships inherit all their properties:

- they are exclusive on their targets but repeatable on their sources: an annotation describes one and only one object even though the number of annotations for any given object may be unbounded;
- they preserve membership: an annotation belongs to a collection if and only if the annotated object does.

Finally, annotation relationships induce a specialization of the notion of collection: A collection is an annotation collection if it is a metadata collection of type IAB. More formally, A is an annotation collection A for C if:

- A is an IAB-collection in the scope of C;
- all the members of A are annotations of members of C; and
- A is an annotation of C.

The definition of annotation relationships and annotation collections – as well as the relationships between these definitions and more generic notions in the gCube Information Model – have been described in 6.1.3.2.

The ABE service and library adopt a model of annotation content that is suitable for exchange of potentially very heterogeneous annotations. In particular, the *exchange model* complements arbitrary, application-specific notions of annotations with a small number of system-level and application-independent properties. The model is defined in terms of XML serialisations of annotations and is in itself an extension of the exchange model adopted by Metadata Management services (cf. Section 6.5.1). In particular, it follows the same design pattern of grouping system-level properties into a *header element* and application-defined properties into a *body element*. The model is formally defined by a pair of XML schemas, where the first schema is an authorised specialisation of the metadata exchange model (within the same namespace) that references annotation-specific elements defined in the second schema (within a dedicated namespace).

6.6.2 The Annotation Back-End Service

The design of the service is distributed across three port-types: the Manager, the Broker, and the Factory port-types.

The **Manager** port-type manages annotations of objects within a single target collection, and thus across all the annotation collections for it. Clients may create annotation collections for the target collection, and then add, update, retrieve or delete annotation from them. These operations map directly onto equivalent operations of the Metadata Management service and are offered here mostly to present a single point of contact for annotations. Clients may also retrieve annotations across multiple annotation collections, though of course no write operation is offered at this level of abstraction. All operations process and produce bulk data to avoid the latency of finer-grained interactions. Similarly, all operations are conceptually overloaded to work either *by-value* – inputs and outputs are entirely included in message payloads – or else *by-reference* – input and outputs are throttled in 'pages' using the resultset abstractions (cf. Section 7.2). Finally, all operations are tolerant to fine-grained failures, through they record them and then report them as return values. Overall, the port-type is the main point of contact for clients that operate within the scope of a single collection C, e.g. wish to present annotations for its objects regardless of the annotation collections in which they are hosted, or else wish to focus their interactions on individual annotation collections.

The Manager port-type is stateful, in that it maintains – both in memory and on the file system – summary information about the annotation collections of the target collections. The information is grouped under local 'proxies' of the target collections, and the proxies are bound to the port-type interface on a per-request basis, in line with the implied resource pattern of WSRF. In particular, the pairing of the Manager interface and collection proxies identifies WS-Resources informally referred to as *Managers*.

Managers are created in response to client requests to the Factory port-type and inherit their scope. It is within this scope that – at creation time – a selection of their state is published in the Information System (cf. Section 5.2). In WSRF terminology, these are the Resource Properties that identify the target collections and their annotation collections and by which Managers can be discovered by ABE clients. Following publication, Managers are kept up-to-date – this time by polling the Information System at regular intervals – with respect to the creation and deletion of annotation collections which do not occur through interaction with the Managers themselves. During their lifetime, Managers satisfy client requests by interacting with WS-Resources and Running Instances of the Metadata Manager service. The interactions are based on instantiations of the Handler's framework included in the gCF, and thus inherit from it best-effort and caching strategies.

The **Broker** port-type manages annotations of objects within either one of multiple target collections $C_1,C_2,\ldots,C_n$. Clients may retrieve annotations for these objects regardless of the collections in which the objects or the annotations are stored. At any point, clients may also narrow their interaction to single target collections by obtaining a reference to corresponding Managers. As for the latter, the operations of the Broker port-type operate in bulk, either by-value or by-reference, and report fine-grained

failures within return values. Overall, the port-type addresses the needs of clients which operate, at least initially, across target collections. For example, the port-type is the recommended point of contact for clients that wish to browse through the annotations of the results of a distributed search.

The Broker port-type is also stateful and its state is an aggregation of the Manager's. In particular the local 'collection proxies' of Managers are grouped in collection sets and then collections sets are bound to the port-type interface into WS-Resources called *Brokers*. Like Managers, Brokers are persisted and publish the identifiers of the target collections in the bound set as a Resource Property. During their lifetime, Brokers also instantiate the gCF's Handler framework to interact with WS-Resources and Running Instances of the Metadata Manager service. The interactions, however, occur indirectly, in that the handlers employed by Brokers rely on those employed by Managers to complete their task.

Finally, the **Factory** port-type creates Managers and Brokers on behalf of clients and is thus stateless.

6.6.3 The Annotation Back-End Library

The library offers the following facilities for interacting with the ABE service:

- it encapsulates standard stub-based interaction behind a local object-oriented interface. Distributed across a set of classes which proxy remote port-types and state abstractions, the interface makes full use of language features – from optimal models of inputs and outputs, to method overloading and parametric type-checking – which cannot be found in the stubs automatically generated from the remote interfaces.
- it offers convenient object bindings for the XML representations of annotations that are required by the remote interface in input and output. The binding occurs within an extensible framework built in turn atop analogous frameworks offered by Metadata and ResultSet libraries. Clients who wish to do so can model annotations as objects with the advantages of expressiveness and early type checking which normally characterise data binding solutions.
- it makes full use of advanced gCF facilities to ensure optimised and best-effort interaction with the running instances of the remote port-types which can be discovered and/or created within the infrastructure.
- it offers built-in support for the linking of annotations of the same object into 'threads' similar to those typically formed by postings in mailing lists and discussion forums, or those associated with chains of versions.
- it provides annotation buffering so as to mediate between the fine-grained nature of operations in interactive sessions and the coarse-grained nature of interaction with the remote port-types. This may enable responsive user interfaces by concentrating bulk network interactions at distinguished 'commit' points which are entirely under client control.

6.7 Data Transformation Service

The gCube Data Transformation subsystem is a group of components responsible for transforming content and metadata among different formats and specifications. The newly introduced gDTS, which is currently under design, absorbes two services formelly offered by gCube under different subsystems. These are the *Metadata Broker Service* which is responsible for transformations of metadata objects and the *Thumbnailer Service* which creates alternative, typically smaller, visual representations of objects. Gradually, these services are incorporated into generic operations offered by the gDTS.

6.7.1 Reference Architecture

gDTS lies on top of Content and Metadata Management services. It interoperates with these components to retrieve Information Objects and store the transformed ones. Transformations can be performed offline and on demand on a single object or on a group of objects.

gDTS has to encompass the functionality already offered by the services it absorbs, i.e. the Metadata Broker and the Thumbnailer Service:

- Transformations of metadata objects can be performed on single record transformations, to bulk by reference input and entire collections. Transformed output can be a snapshot of its input or a mirror continuingly updated along with its original source. Individual transformation programs are treated as building blocks to synthesize more elaborate and complicated new programs.
- Thumbnails are in general created on demand. This means that they are not pre-produced and stored in CMS, but they are produced when somebody is asking for them, as for example in the case of a user browsing the content stored in a repository through the portal. When the input is an image, it creates a thumbnail of the image, given the desired resolution. For any other type of digital object, it returns a predefined icon that specifies the type of the content.

In order to achieve this, gDTS employs a variety of pluggable converters to transform digital objects between arbitrary content types, and takes advantage of extended Information on the content types to achieve the selection of the appropriate conversion elements. Currently, gCube Data Transformation Service provides only support for transformations of metadata objects and generation of thumbnails, nevertheless due to the pluggability of new functionality it is capable of handling any type of content.

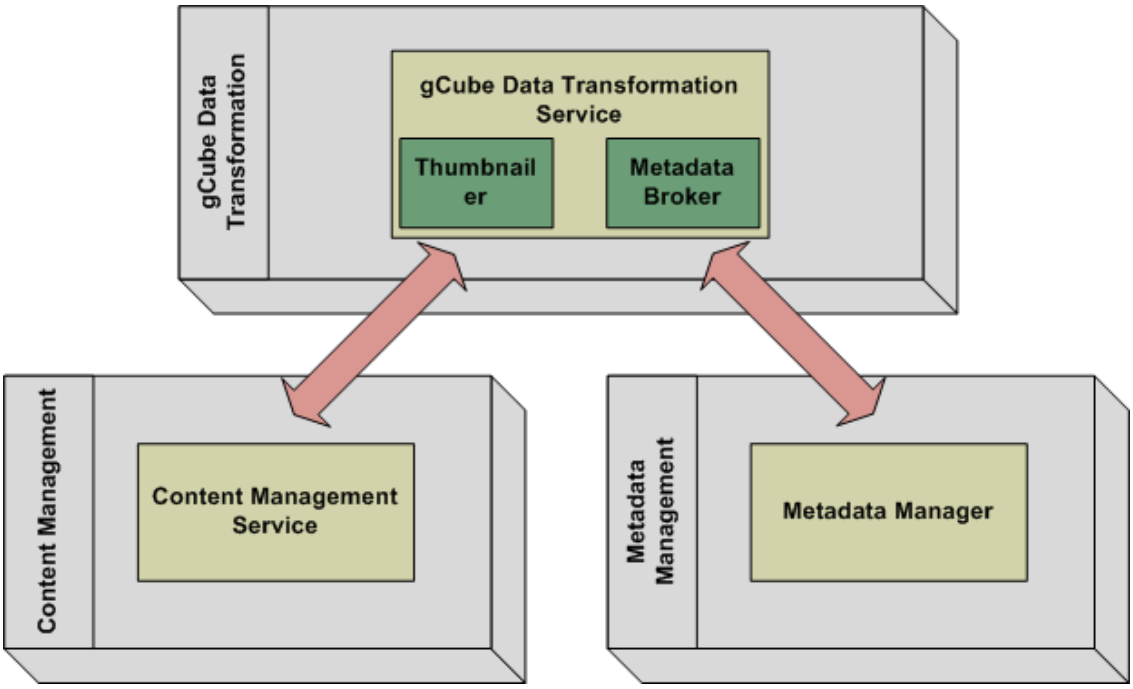

Figure 33. Data Transformation Services in gCube

As already mentioned, the main functionality of the gCube Data Transformation Service is to convert digital objects from one content format to another. The conversions will be performed by transformation programs which either have been previously defined and stored or are composed on-the-fly during the transformation process. Every transformation program (except for those which are composed on-the-fly) is stored in the IS

The gCube Data Transformation Service is presumed to offer a lot of benefits in many aspects of gCube. Presentation layer benefits from the production of alternative representations of multimedia documents. Generation of thumbnails, transformations of objects to specific formats that are required by some presentation applications and projection of multimedia files with variable quality/bitrate are just some examples of

useful transformations over multimedia documents. In addition, as conversion tool for textual documents, it will offer online projection of documents in html format and moreover any other downloadable formats as pdf or ps. Annotation UI (cf. Section 8.4.8) can be implemented more straightforward on selected logical groups of content types (e.g. images) without caring about the details of the content and the support offered by the browsers. Finally, by utilizing the functionality of Metadata Broker, homogenization of metadata with variable schemas can be achieved.

The transformations can be performed on single content objects or even collections of them, which can be retrieved from the content management service. Similarly, metadata records and metadata collections are retrieved by the metadata manager. Furthermore, binary content from external sources can be input of gDTS in form of result set with blob elements.

In order for gDTS to be invoked, apart from the input and output, the clients should indicate information about what transformation will be performed. This can be done in two ways. Either by explicitly setting the transformation program which will perform the transformation or implicitly by indicating the source and target content type. In the second occasion, the service is responsible to find the appropriate transformation program.

In gDTS the management of the content types conforms to the MIME specification (RFC 2045, 2046), in order to be compliant with the other mime implementations as browsers, mail clients, etc. In this context, a digital object's content type is defined by the media type, the subtype identifier plus a set of parameters, specified in an attribute=value notation.

6.7.1.1 Elements of gDTS

The gCube Data Transformation Service consists of a variety of components which collaborate in order to achieve the desired functionality. These are components which act as communication adaptors and are responsible to contact content and metadata management services (*Input* and *Output Handlers*) as well as the information service (*IS Retriever*), but the main part of the system is called the "core" (g*DTS Core*) and it comprises of elements that are responsible to determine and perform the requested transformations.

Depending on the requested input and output procedures Input and Output Handlers perform the invocations to retrieve and store content respectively, from the other information management services i.e. CMS and MMS. Apart from the content, information about its format, if available, is fetched by the Input Handler. The content and format info are passed to DTS Core in a form of abstract input the Input Source. Accordingly, Output Handler gets from DTS Core the transformed content in a Data Sink. Input and Output Handlers are also responsible to maintain relationships between the original and the transformed content.

gDTS Core receives the Input Source produced by the Input Handler as well as the information about the requested transformation. Transformations are performed by transformation programs. The clients may specify which Transformation Program will be used or alternatively let the service to determine it.

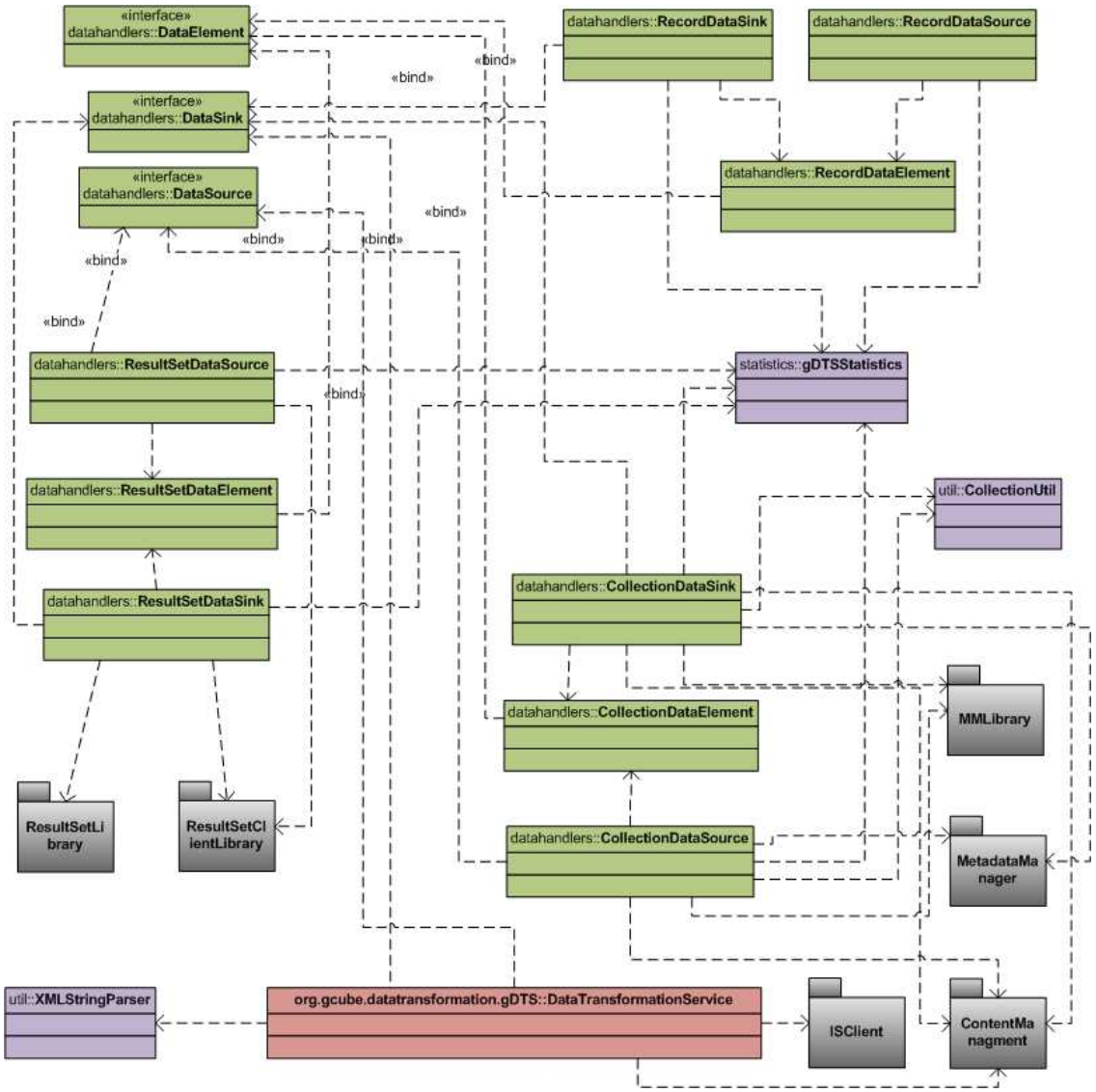

Figure 34. gDTS Internal Component Diagram

6.7.1.1.1 Transformation Programs and Transformation Rules

A *Transformation Program* is an XML document describing a complex transformation from a source format to a target format. Each transformation program describes the way data should be transformed, but does not contain the data itself. So a transformation program is not bound to some specific data but can be applied to any data source, as long as the data is in the format expected by the transformation program. Each transformation program can reference other transformation programs and use them as "black-box" components in the transformation process.

Transformation programs consist of the following entities:

- One or more data inputs. Each data input defines the mime type, the format parameters and type (object, collection or ResultSet) that the data that will be mapped to it (when this transformation program is used) should have.
- One or more input variables, which are just variable string elements whose value is given by the caller when the transformation program is about to be executed. They can be thought of as parameters required by the transformation program.
- Exactly one data output, which defines the target mime type, format parameters and type of the output data produced by the transformation program.

- One or more *Transformation Rules*, which define simple transformations. Transformation rules are the building blocks of Transformation Programs. They work together in order to perform the complex transformation defined by the transformation program that contains them. Each transformation rule is actually a XML description of the interface (inputs and outputs) of the underlying java class (Program) which is responsible for performing the actual transformation on the input data. So a transformation rule can be thought of as a wrapper for a Program. Programs are described later in this document.

Transformation rules consist of:

- A declaration that specifies the java class (program) that this transformation rule uses.
- One or more data inputs, just like transformation programs. Each data input describes the data that should be passed to the underlying java program. The output produced by other transformation rules can be mapped to a transformation rule's input, thus forming a chain of data processing elements.
- Exactly one data output, which describes the format of the data produced by the transformation rule and also acts as a placeholder for the actual data. The output of the whole transformation program is always set to the output of the last transformation rule contained in it.

In Figure 35, an example of a transformation program is presented. Transformation program 1 is composed of two transformation rules. Transformation rule 1 just invokes transformation program 2, which in this context is just used as a "black-box" component. The output produced by transformation program 2 becomes the output of transformation rule 1, which is then mapped to the input of transformation rule 2. Finally, the output produced by transformation rule 2 is the whole transformation program's output.

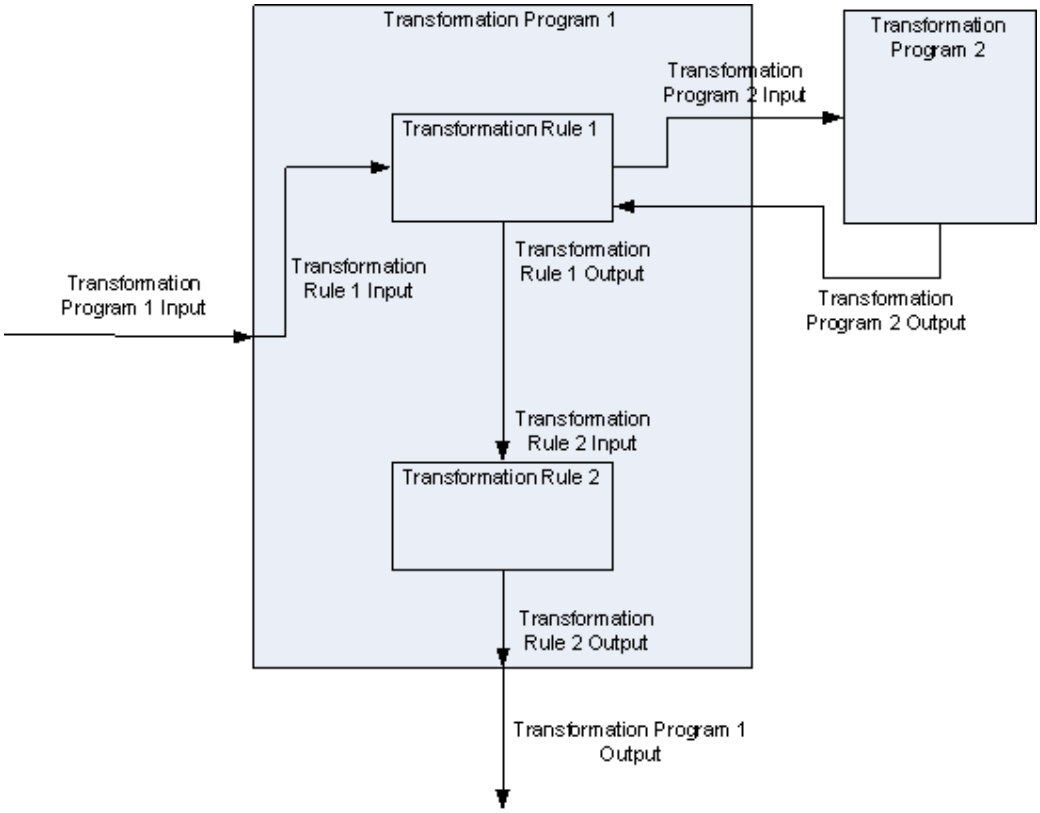

Figure 35. Example Transformation Program

The transformation programs used formerly by the Metadata Broker, called "Broker Transformation Programms" are a subset of these maintained by gDTS. gDTS operates over transformation programs with variable mime types and format parameters, when the Metadata Broker deals with programs transforming documents with mimetype text/xml converted to text/xml and with format parameters only the schema and the language.

6.7.1.1.2 Transformations Graph and Composition of Transformation Programs

gDTS maintains a Transformations Graph which is responsible to find transformation programs which can be used in order to perform a requested transformation. It is also used to compose a new transformation program consisting of more than one transformation rules. The Transformations Graph consists of nodes which correspond to object formats and the edges to transformation programs. So, when a request is made to transform an object from one source format to the target, gDTS utilizes the information contained to the graph to find one or more transformation paths i.e. sequences of transformation rules.

Then, depending on the clients' request, the service may return all the available transformation programs, existing or composed, to the client and let him choose which one he wants to use or perform the transformation by selecting one of them. In the second case, the choice for which transformation program will be used is based on variable factors. The current implementation bases the selection of the program to the following rules. If there is an existing program that supports the transformation with all the source and format parameters then this is used. Else, the first of the available composed program that supports the transformation is used. Finally, if there is no transformation program available that can perform the transformation taking into consideration all the format parameters, a generic one will be used. Generic transformation program is the one that can perform the source mimetype to target mimetype transformation i.e. format parameters are not examined.

The next step regarding the selection of the transformation program automatically by the gDTS, is to let the user define preferences concerning the performance or the quality of the transformation. By keeping the appropriate statistics and metrics, the service will be able to base its selection on the client's requirements.

6.7.1.1.3 Programs

The available transformations that the gDTS can use reside externally to the service, as separate Java classes called *Programs* (not to be confused with 'Transformation Programs'). Each program is an independent, self-describing entity that encapsulates the logic of the transformation process it performs. Every time a transformation request is received, the gDTS starts executing sequentially the transformation rules that compose the given transformation program. Each transformation rule references a java program that is responsible for performing the actual transformation. The gDTS loads these required programs dynamically as the execution proceeds and supplies them with the input data that must be transformed. Since the loading is done at run-time, extending the gDTS transformation capabilities by adding programs is a trivial task. The new program has to be written as a java class and referenced in the classpath variable, so that it can be located when required.

The gDTS provides helper functionality to simplify the creation of new programs. This functionality is exposed to the program author through a set of java classes, which compose the *gCube Data Transformation Library.* This functionality is based on the concept of generic *data sources* and *data sinks*. Whenever the gDTS is invoked, the caller-supplied data is automatically wrapped in a data source object. In a similar way, the output of the transformation is wrapped in a data sink object. The source and sink objects can then be used by the invoked java program in order to read each source object sequentially and write its transformed counterpart to the destination. This processing of data objects is done homogenously because of the abstraction provided by

the data sources and data sinks, no matter what the nature of the original source and destination is (object collection or ResultSet).

Programs may be by themselves converters but also they may employ command line programs, XSLTs or other external services to perform the transformations. In this way, the services provided by gDTS for converting digital objects can be easily enriched to cover a great amount of requirements.

6.7.1.1.4 Persistent transformations

Each call to the gDTS describes a request for content transformation. Such a transformation process is not required to maintain a state, unless it is needed to automatically update the output whenever there is a change on the input(s). Naturally, this feature is only available to transformations applied on content and metadata collections, generating new ones. In the case of persistent transformations, information about the transformation is stored in the Content Management layer, inside a collection created for this purpose. This collection contains one Information Object for every collection involved in a persistent transformation. Each such Information Object contains information about every active persistent transformation requested on the collection that this Information Object refers to. Apart from the data stored in the Content Management layer, no other method is used to maintain the state of the service, such as WS-Resources.

Whenever one or more objects are added to/deleted from/updated in a collection, the gDTS is notified by the appropriate service. Then, the persistent transformation state is retrieved from the Content Management Layer in order to check if the modification should be reflected on any derived collections, generated using the gDTS in the past. If it should, then the transformation which was originally used to produce the derived collection is executed once again to transform the modified objects.

6.7.1.1.5 Performance Statistics

The gCube Data Transformation Library provides a mechanism for keeping various performance statistics about the gDTS. This mechanism must be initialized by defining *performance metrics*. Each performance metric represents an important common operation whose performance needs to be monitored and is defined simply by a name and a description. For instance, a performance metric may be called "TimeToTransformMetadataRecord" and represents the time it takes for the gDTS to transform a single metadata record. After all the metrics have been defined, the gDTS can start keeping statistics. Each time a transformation is requested, the related program is responsible to keep track of the time required for each common operation (e.g. to transform a single metadata record). When the operation completes, the program provides the measured time values to the statistics mechanism, using the associated metric name. This way, the statistics mechanism keeps track of the minimum, maximum and mean value for each metric.

Every five minutes, the performance statistics are published on the IS as part of the service's running instance profile, in XML format. This is done by a background thread which is spawned when the service starts running. This way, the performance of the gDTS can be monitored by inspecting the published values and possible bottlenecks can be identified. Of course, the statistics mechanism is extensible, meaning that each program can define its own performance metrics. There is no restriction on the number of metrics that can be monitored.

6.7.2 Resources and Properties

Currently the gDTS is designed as a stateless WSRF service, thus it exposes no WSRF resources. In order to take advantange of Content Management Service, its "state" is managed by CMS constructs. However in subsequent stages of the design gDTS will support also WSRF statefull operations.

Resources posted and consumed via the DIS are the Transformation Programs.

6.7.3 Functions

- **transform()**: which receives the Information Object, a collection or a list of objects and the source and target formats and the transformation path and performs the requested transformation, returning or persisting the results appropriatelly.
- **calculateTransformationPath()**: which receives the Information Object or a source format, along with a target format and identifies a list of transformation paths (or the best of them) for applying a transformation.
- **identifySourceFormat()**: which receives an Information Object and identifies its content type details, depending on CMS published information and potentially in object payload.

7 THE INFORMATION RETRIEVAL SERVICES HIGH-LEVEL DESIGN

The gCube Information Retrieval Services is a family of application-specific components which build upon the rest of gCube services. The IR services focus on the business logic, more accurately Information Retrieval. The IR family of services can be decomposed in three major categories: Search Framework, Index Management Framework and finally Distributed Information Retrieval Support Framework. Each framework is focused on a specific area of Information Retrieval, but works in synergy with services from the other two frameworks, as well as with services belonging to different subsystems, such as infrastructural ones, in order to provide a rich set of IR operations.

First, the overall architecture of the IR services (basic subsystems and their integration patterns) is presented and subsequently each prime IR component is analyzed, emphasizing on its inner functionality and its interaction with its working environment.

7.1 Overall Architecture

Figure 36 depicts the various components which participate in the IR procedure, though not all defined in the IR Service Group. They are divided in four groups: the Search Orchestration group, Process Execution, IR service group and Storage Layer. External components which do not belong to the IR family, but rather have an association to one or more IR components are also included. These are the Portal component (cf. Section 8), Information System (cf. Section 5.2), Process Management Service (cf. Section 5.6), and finally the Storage Management (cf. Section 6.3).

The Search Orchestration group includes the components responsible for managing the search procedure. In more detail, this group receives search activities (queries) from the Presentation Layer (search portlet), initiates the query processing procedure, forward the execution plan to satisfy the activity and finally forward the final results back to the Presentation Layer.

The Process Execution group, in the IR context, provides the mechanics for executing user or machine-generated processes. These processes include online, machine-generated processes, mainly participating in search activities as execution plans corresponding to user queries, or "continuous", background, user-defined processes needed for various system-oriented, housekeeping activities, such as index feeding. Process Execution is not an "official" part of the IR family but plays major role since it accommodates its functionality.

The IR Service group includes components which perform the actual IR activities, either offering information retrieval or providing some information processing facilities. The term "*information*" is intentionally used instead of "*data*" to emphasize on the fact that IR components handle contextual pieces of data and not plain, semantically undefined and potentially unstructured data, which cannot be used in the IR context. Thus, components are specifically designed to work in synergy so as to provide a rich, full-fledged set of IR functionalities. To some extend, they employ lower level, data management systems, such as the Storage Layer, mostly as state and data backup mechanism, but in principal they are treated as stand-alone providers of higher level IR services to the end user.

The grouping model adopted is not directly mapped to the three major IR frameworks, but rather presents an activity-oriented representation of IR in gCube. The actual IR frameworks consist of components that are spread around in these groups. The Search framework includes components from all three groups. The Index and DIR Support framework components lie exclusively in the IR Service group. In the following paragraphs each framework is presented distinctly, its constituent components are described and their shared, cooperative execution context, is analysed delving into the business logic and inner mechanics.

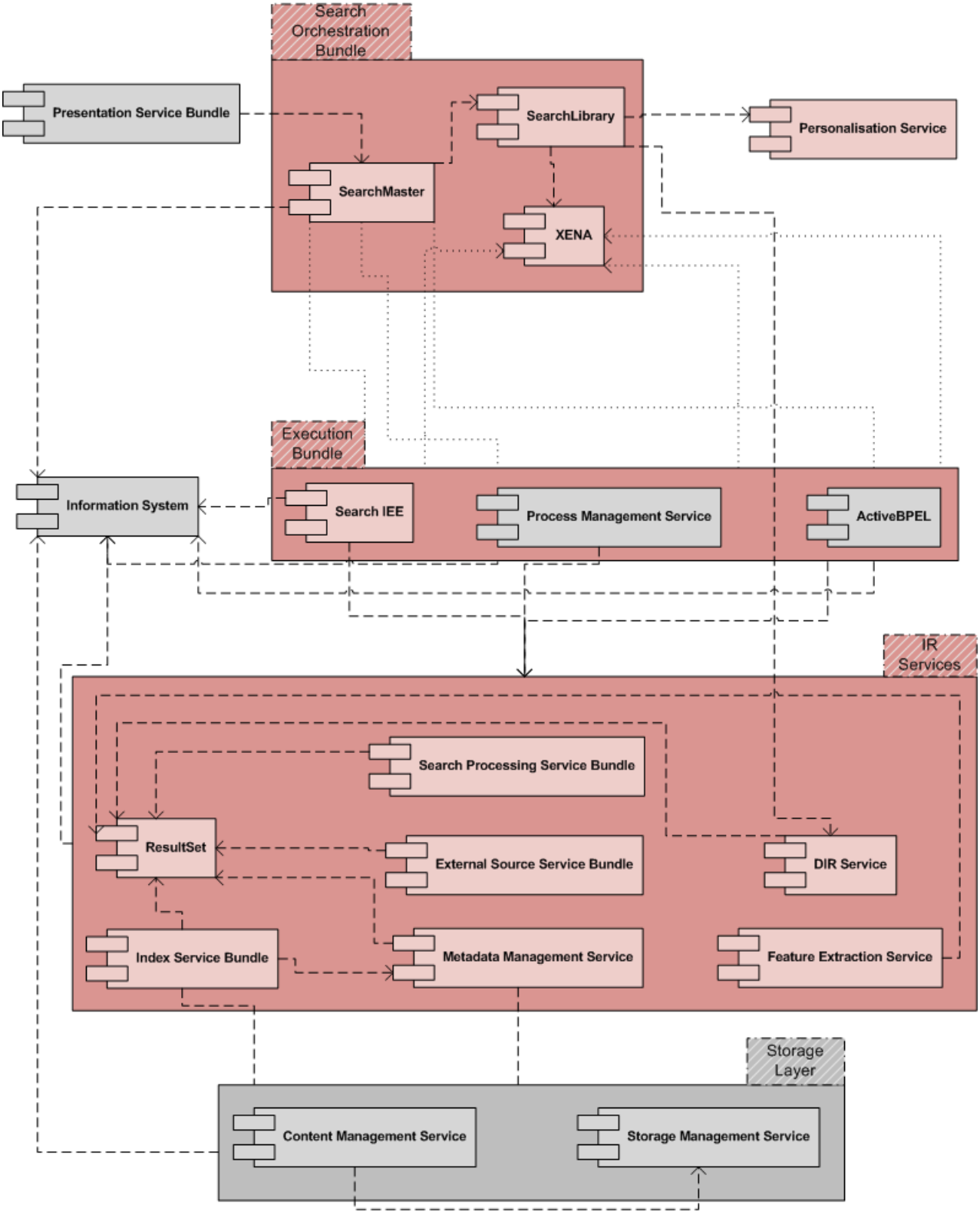

Figure 36. Components Employed in IR

7.2 gCube ResultSet (gRS)

Various components in the overall infrastructure need to communicate data in an efficient way allowing maximum performance and yet not deviating from the standards followed during the design process. To manage the data transfers of interacting components the *gCube ResultSet* mechanism is put into use in many situations. For example, the intermediate steps towards producing the final search output are handled by *SearchOperator* services. These include Fielded Search operators, FullText Search operators, ForwardIndex Search operators, result Sorting operators and many more that are analyzed in other sections. All these operators make use of the *gRS* mechanism to receive their input and send output data to other services.

7.2.1 Reference Architecture

One fundamental operation in several gCube services is the production and consumption of data sets. These data sets refer to Information Objects and their metadata, either derived on-the-fly or pre-existing. The operation call chain (or workflow) to be constructed is not predefined and yet services have to be able to exchange data in a standardized manner. This standardized manner of communicating results between various operators is covered through the use of the *ResultSetService*. This service is a WSRF compliant Web Service. The producing operators construct gRSes in which they insert the results of their execution. At some point these producers wrap the *gRS* in a WS-Resource and communicate the End Point Reference of this resource to any operators waiting for this output.

It is not desirable to fix the structure of the data, since this would severely limit the generality of the data and metadata processing. However, these data have a special minimal meaning: they refer to Information Objects and they have the nature of the record, yet the structure inside a record is expected to be semi-structured, i.e. not explicitly fielded. Thus, there is a need for a vehicle that will act as a unifying frame around data to be exchanged in between services, adding minimal structural requirements with regards to the ones raised by the application domain in target (Service Oriented Distributed Query Processing). This vehicle is the *ResultSet* Element, and is kept in a soft state from the *ResultSetService* WS-Resource.

Figure 37 shows the most important classes of the *gRS*. The *ResultSetService* makes use of the *ResultSetLib* and any *ResultSet* implements a Handler interface. The *ResultHandler Interface* depicted, is a logical interface to show that elements share a set of methods. Their implementations depend on the referenced element. In the *ResultSet*, the full implementation of the functionality is needed. In the *ResultSetService*, the method's implementation is solely a forwarding of the request to the underlying *ResultSet* element.

The ResultSet class is the object that will carry back the results of an operation. Not only will it be used to return results to the user but also in order to transfer intermediate results from one Operation invocation to another. It is expected to be one of the most common operands for QueryPlan invocations.

The ResultSet class inherits functionality from the Information Object class, so that it can be stored by relying on the facilities dedicated to these objects. It contains items of ResultSetEntry, which are actually the records of the result set. Each record has a unique identifier and a pointer to the object it is representing (multiple references to the same object might be allowed). The rest of the fields of the record might not be flat but present a tree-structure (XML like) and will be typically used for carrying metadata of the referenced item.

The ResultSet class provides operations for record movement (Movefirst, MoveNext, EOF etc), retrieving/refreshing next chunk of records (Retrieve, Refresh etc), a placeholder for the originating query and the context of the execution, etc.

Some options for the realization for the ResultSet include:

- The provision of WS-Resource wrapped around a ResultSet persistent object.

- The creation of proxy class ResultSetRef that can provide a generic mean for accessing a ResultSet, be it a WS-Resource, a logical file name or a ResultSet object.

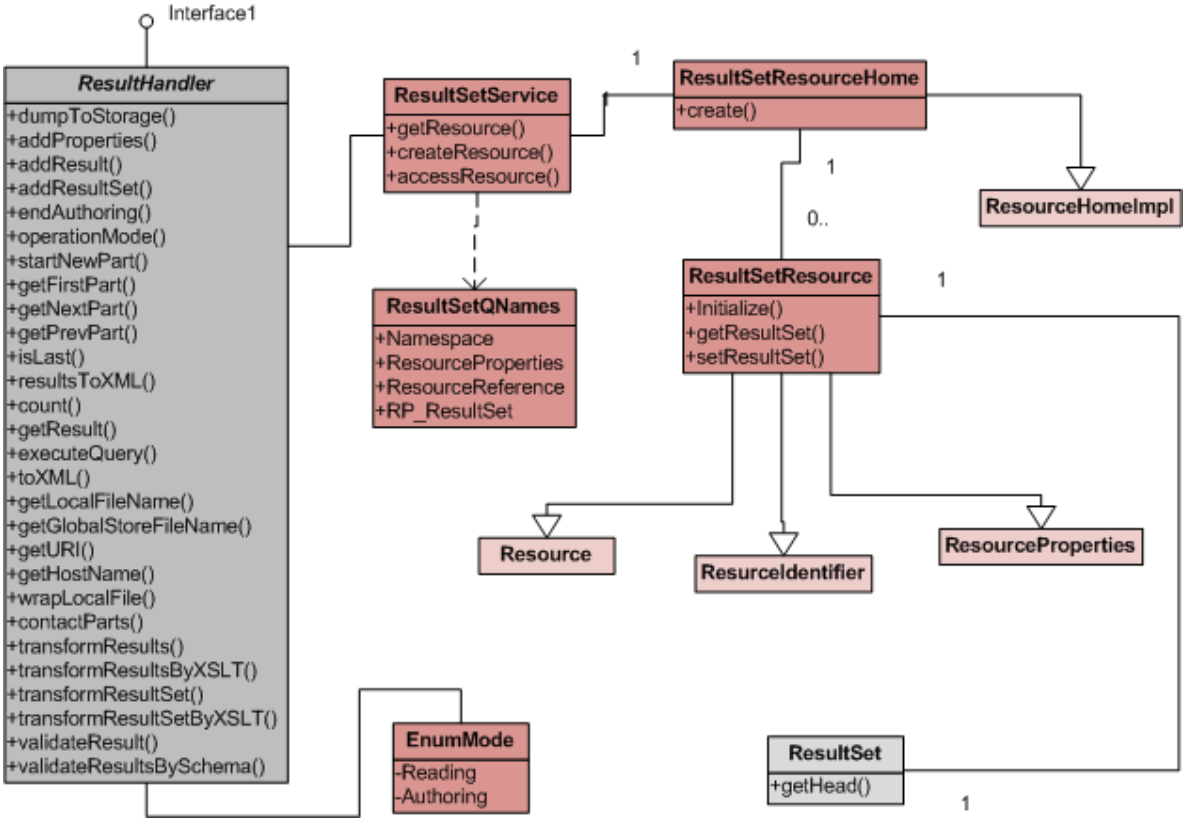

Figure 37. ResultSet Class Diagram

7.2.2 Resources and Properties

The managed resource of the *ResultSetService*, the *ResultSet* Element, can efficiently be updated, transformed, and retrieved as the output produced by a service. Since this Element is needed by many components, it is expected to be part of any gHN. The *ResultSet* Element is expected to have a very brief lifetime and a new instance of it is created every time a producer needs to output its results, or a consumer needs to access or modify an existing resource. For this reason the *ResultSetService* resources, the ResultSet elements, are not persisted. This doesn't mean that the intermediate results of a data transfer are not placed in permanent storage. Intermediate results are handled by the ResultSet element and are organized as a chain of parts placed in permanent storage locally. These parts, though, would be unreachable, if an access point was not provided. The ResultSetService provides that access point. Any ResultSet element can be reached by the use of an identifier. This identifier in the case of the ResultSetService is a WS-Resource end point reference. This resource ResultSet element, though, is not persisted; it is kept as in-memory state handling the persisted payload of the result set.

7.2.3 Functions

The following functions are among those that clarify the operation of result set:

- **startNewPart()**: closes a "page" of results and starts a new one so that the previous can be posted to the consumer.
- **addResult():** adds a record to the ResultSet.
- **endAuthoring()**: stops the production of a resultset.
- **Islast()**: a close analog to EOF, referring to parts of a ResultSet.

- **toXML()**: Returns the raw payload of the gRS.
- **transformResultsXXXX()**: applies XSLT transformations to rows or the entire resultset
- **dumpToStorage():**essentially serialises a resultset to the File System
- **wrapLocalFile():** essentially deserialises a resultset from the File System

7.3 Search Framework

The Search Service is the element of gCube responsible for providing search capabilities. It receives a user query, potentially along with supplementary information (such as a document to be used for content-based search), produces an appropriate execution plan to compute this query, feeds that plan to an execution engine and finally forwards the final results to the end user.

7.3.1 Reference Architecture

In order to obtain an in-depth understanding of the Search Framework, we present the following two diagrams are given:

- An operational diagram which provides a high-level representation of the Search Framework operational context, along with the cooperating external components.
- A class diagram which provides a more in-depth representation of the major classes that are part of in the Search Framework.

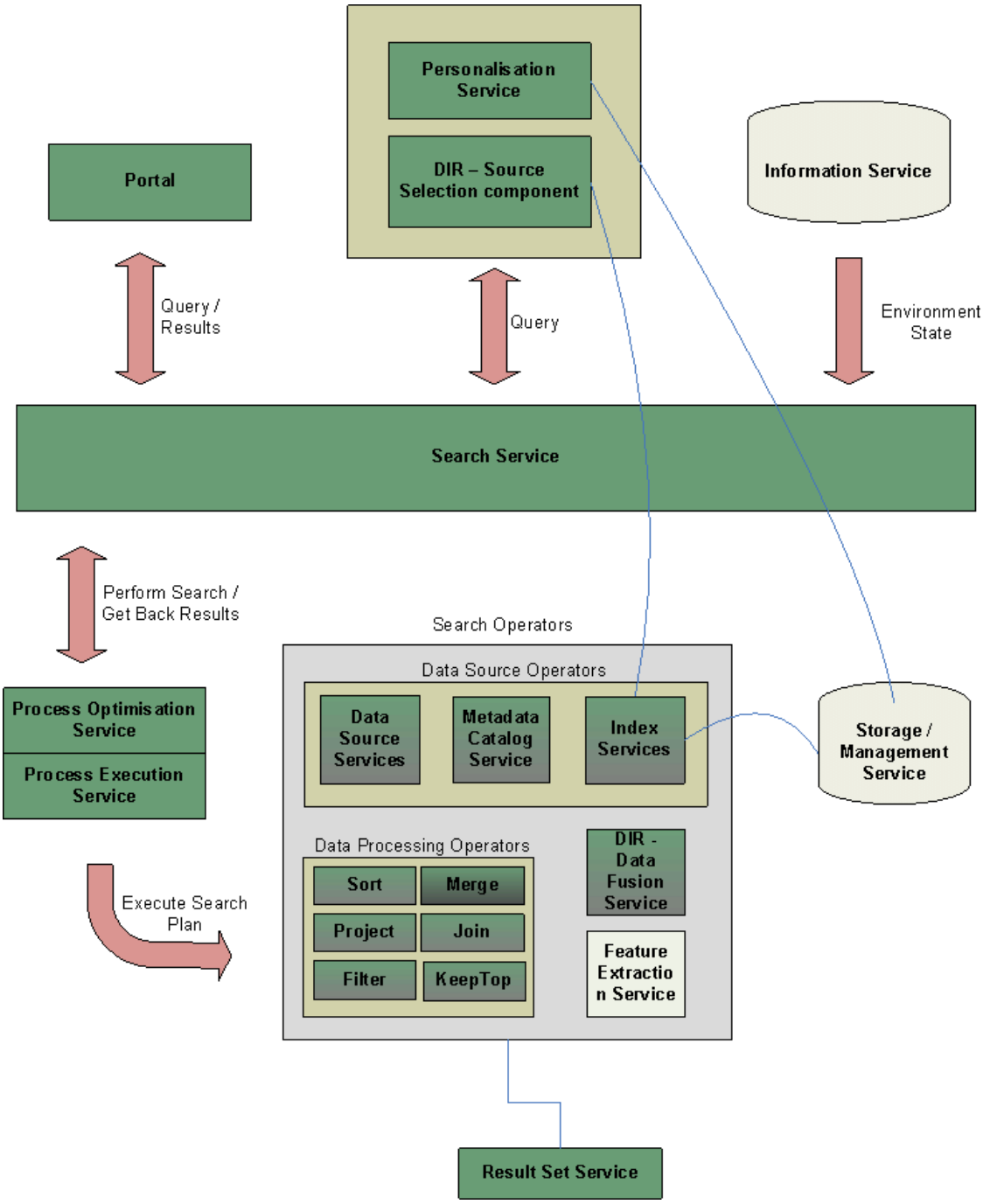

Figure 38. Search Operational Diagram

In the above diagram thematically related services are grouped in bounding boxes. The main flows of information (and control) are drawn with heavy arrows while secondary flows are drawn with lines

Search service receives the user Query is constructed by the user interface and it is expressed in terms of XML Infoset. Additionally a query language[10] has been adopted and semantically defined to provide textual means of specifying user queries.

[10] The gCube Quary Language is presented in Appendix A.

After accepting the query, the Search Service has to collect system information regarding its resources and their status. This information is employed by the Search Service in order to produce a way to compute the query in an (near) optimal manner, without any sacrifice to its semantics.

At this point, the system can actually create a plan for executing the query. However the gCube Search Service adds an advanced Query-Preprocessing layer that comprehends and modifies the query in order to improve user experience in various terms. The two major query preprocessors in gCube are the Query Personalisation (cf. Section 7.5) and the DIR (cf. Section 7.6). The former is responsible for modifying the query in order to satisfy the user preferences that have been recorded in his/her profile. The later attempts to filter out content collections that do not seem to contain information relevant to the user query, so that the system does not consume any resources for a near meaningless operation. Linguistic processing (such as stemming) or using ontologies to expand search terms are also examples of such preprocessing. Although additional preprocessing steps are supported, they are not part of the major gCube implementation and they can be provided externally as pluggable components.

After the query is finally formed by the preprocessors, the system has to construct a plan in order to carry out its execution. In the case of a user posting a complex query, its execution/computation might involve a lengthy series of steps and there might be a large number of alternative means for achieving its target. For example, semantically equivalent series of operations might exit or multiple alternative implementations and/or instances of services might be available in a particular (sub)VO. Thus the system has to find a way via its planner so that the query is served "optimally".

The outcome of the planner is the query execution plan, which is a workflow of the so called SearchOperators. SearchOperators are web services that have a particular highly focused task to carry out. Examples of operators are sorting on XML results, looking up similar images based on a provided prototype, performing XPath queries on XML data, merging results coming out of different collections, etc.

Among these operators, the *Index lookup (cf. Section 7.3.4.9)*, the *DIR/DataFusion (cf. Section 7.3.4.14,)* and the *Feature Extraction (cf. Section 7.3.4.12)* are quite special. Index lookups are made possible through providing a mechanism that creates and maintains these constructs in an efficient manner, providing index-specific matching operations. DataFusion (*cf. Section 7.6*) consolidates information on the result sources in order to achieve merging several ranked result lists into one meaningful ranked list. Feature Extraction utilises complex computationally intensive algorithms to calculate indicators and extract features of raw data, in order to facilitate content based search (e.g. search for images similar to a prototype etc).

Although the Search Service constructs the workflow of operations, it is not responsible for performing the actual execution. The workflow is authored in a standard language (BPEL) and is submitted to the dedicated execution mechanism of gCube, which is provided by the Process Management services (*cf. Section* 5.6).

The last operation of this workflow will be to return the results to the workflow originator, i.e. the Search Service. Finally the Search Service will deliver the results back to its requestor.

The Search Service is a logical abstraction of the search component group that can be decomposed in three major categories.

In the following diagram the most important classes and associations, regarding the Search Framework are presented.

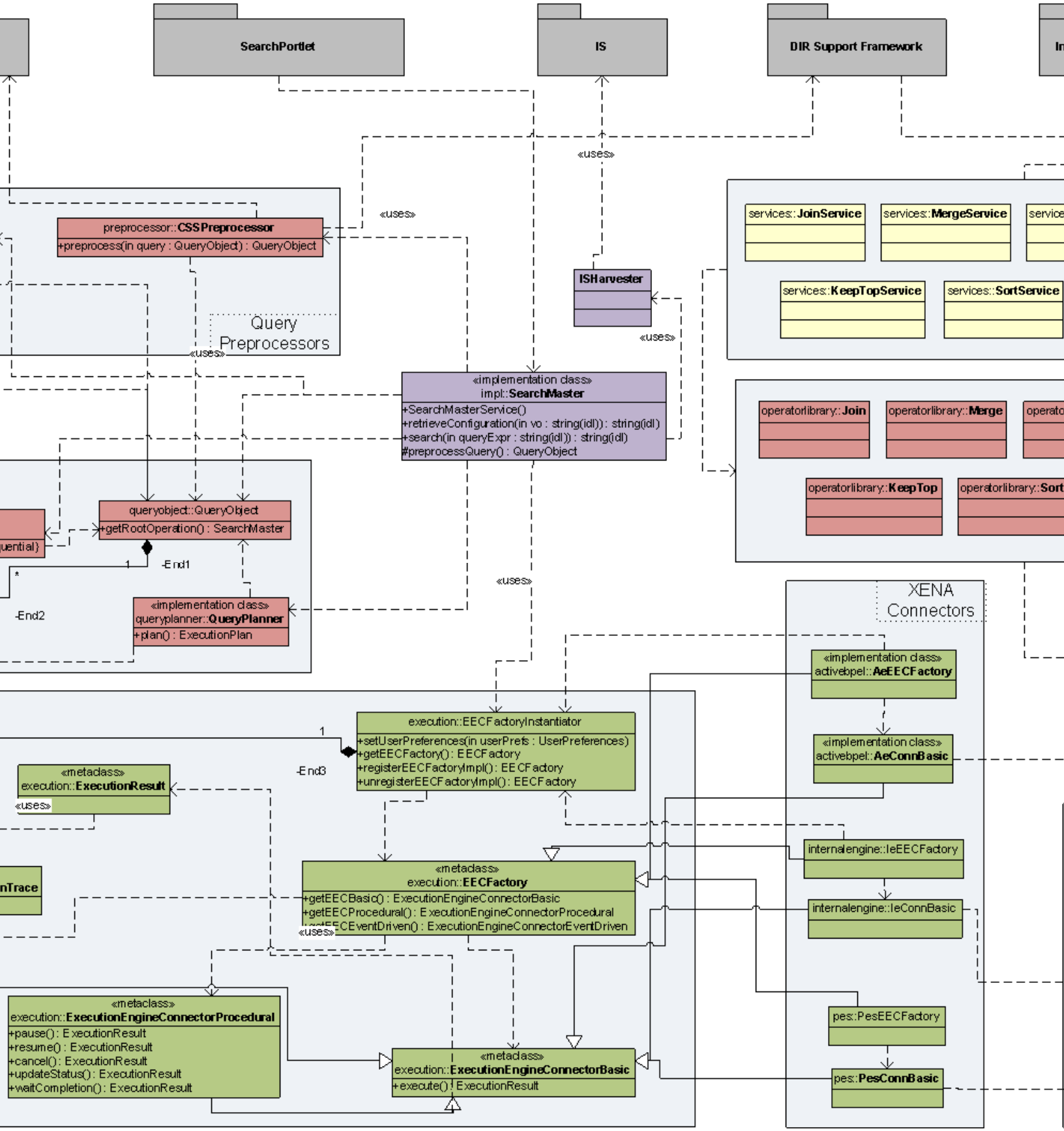

Figure 39. Search Framework

The Search Framework consists of three major component categories: *SearchMasterService*, *SearchLibrary* and *SearchOperators*. The first two categories are atomic in the sense that they consist of a single entity. The third one is a family of *gCube* services which expose various search functionalities.

The *SearchMasterService* provides the access point to the gCube Search Engine. It receives a user query from the *Portal*, and along with some environment information received from the *Information Service* (*IS*), it initiates the query processing procedure. This procedure comes up with an *ExecutionPlan* which is fed the appropriate *ExecutionEngineConnector* which, in turn, forwards it to its corresponding execution engine (external to the Search Framework).

The *SearchLibrary* is an all-in-one bundle that incorporates all the query processing, as well as the actual implementation of the *SearchOperators*. The most critical subcomponents are: *Query Processor Bundle* (*QueryObject*, *QueryParser*, *QueryPlanner*, *QueryOptimizer)* (green coloured), *Search Operator Library* and the *eXecution ENgine API* (*XENA*). These components are analysed in the *SearchLibrary* section (*cf. Section* 7.3.3).

Finally, the *SearchOperators* bundle is decomposed into several physical packages offering low level data processing elements. It is a family of gCube services, each of which is dedicated to a specific data processing facility. As previously mentioned, the *SearchOperators* bundle is a thin wrapper over the *SearchLibrary* which provides the actual implementation.

In the following paragraphs these three components are presented along with their constituent *subcomponents* and their inner mechanics and relationships to the rest of the gCore/gCube context.

7.3.2 SearchMasterService

The SearchMasterService is the entry point to the functionality of the search engine. It contains the elements that will organize the execution of the search operators for the various tasks the Search engine is responsible for.

There are two steps in achieving the necessary Query processing before it can be forwarded to the Search engine for the actual execution and evaluation of the net results. The first step is the transformation of the abstract Query terms in a series of WS invocations. The output of this step is an enriched execution plan mapping the abstract Query to a workflow of service invocations. These invocations are calls to Search Service operators providing basic functionality called *SearchOperators*. The second step is the optimization of the calculated execution plan.

The *SearchMasterService* is responsible for the first stage of query processing. This stage produces a query execution plan, which in the *gCube* implementation is a directed acyclic graph of *SearchOperator* invocations. This element is responsible for gathering the whole set of information that is expected to be needed by the various search services and provides it as context to the processed query. In this manner, delays for gathering info at the various services are significantly reduced and assist responsiveness.

The information gathered is produced by various components or services of the *gCube* Infrastructure. They include the *gCube Information Service* (IS), *Content and Metadata Management*, *Indexing service* etc. The process of gathering all needed information proves to be very time consuming. To this end, the *SearchMasterServic*e keeps a cache of previously discovered information and state.

The *SearchMasterService* validates the received *Query* using *SearchLibrary* elements. It validates the user supplied query against the elements of the specific Digital Library Instance. This ensures that content collections are available, metadata elements (e.g. fields) are present, operators (i.e. *services*) are accessible etc. Afterwards it performs a number of preprocessing steps invoking functionality offered by services such as the

Query Personalisation and the *DIR (§7.6)* (former *Content Source Selection*) service, in order to refine the context of the search or inject extra information at the query. These are specializations of the general *QueryPreprocessor* Element. An order of *QueryPreprocessor* calls is necessary in the case where they might inject conflicting information. Otherwise, a method for weighting the source of the conflicting information importance is necessary. Furthermore, a number of exceptions may occur during the operation of a preprocessor, as during the normal flow of any operation. The difference is that, although useful in the context of gCube, preprocessors are not necessary for a Search execution. So errors during Query Preprocessing must not stop the search flow of execution.

The above statement is a sub case of the more general need of a framework for defining fatal errors and warnings. During the entire Search operation a number of errors and/or warnings may *emerge*. Depending on the context in which they appear, they may have great or no significance to the overall progress of execution. Currently, these cases are handled separately but a uniform management may come into play as the specifications of each service's needs in the grand plan of the execution become more apparent at a low enough level of detail.

After the above pre-processing steps are completed successfully, the *SearchMasterService* dispatches a *QueryPlanner* thread to create the *QueryExecutionPlan*. Its job is firstly to map the provided *Query* that has been enriched by the preprocessors to a concrete workflow of WS invocations. Subsequently, the *QueryPlanner* uses the information encapsulated with the provided *Query*, the information gathered by the *SearchMasterService* for the available *gCube* environment and a set of transformation rules to come up with a near optimal plan. When certain conditions are met (e.g. the *QueryPlanner* has finished, time has elapsed, all plans have been evaluated), the planer returns to its caller the best plan calculated. If more than one *QueryPlanners* are utilized, the plans calculated by each *QueryPlanner* are gathered by the *SearchMaster*. He then chooses the overall optimal plan and passes it to a suitable execution engine*,* where execution and scheduling is being achieved in a generic manner. The actual integration with the available execution engines and the formalization of their interaction with the *SearchMasterService* is accomplished through the introduction of the *eXecution Engine Api* (*XENA*), which is thoroughly analyzed in the SearchLibrary section (*cf. Section* 7.3.3). In this formal methodology, the *SearchMasterService* is able to selected among the various available engines, such as the ProcessExecutionService, the *InternalExecutionEngine* (see 7.3.3.4) or any other WS-workflow engine. These engines are free to enforce their own optimization strategies, as long as they respect to the semantic invariants dictated by the original *ExecutionPlan*.

The following diagram depicts the component's internal design.

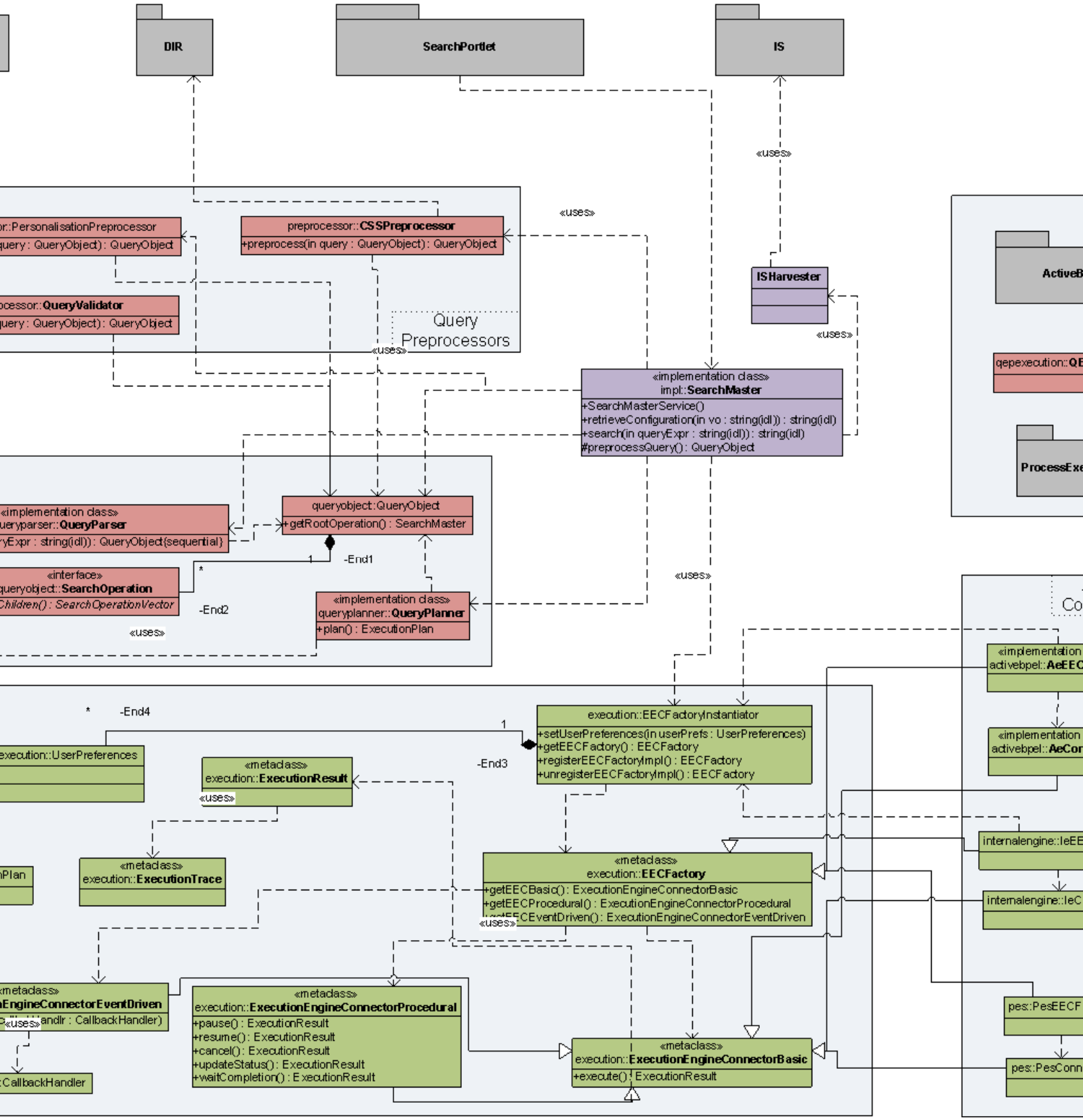

Figure 40. Search Master Architecture

Finally, the *SearchMasterService* receives the final *ResultSet* from the execution engine and *pass* its end point reference back to the requestor.

7.3.2.1.1 Resource Information Handling

The resources handled by the *SearchMasterService* are the Resources gathered by the *IS* and preserved by the *SearchMasterService* as cache. The cache, which is preserved between consequent calls by the *SearchMasterService*, keeping track of its freshness and periodically or on demand updating it, is considered to be service configuration information. Thus it will not be handled using the WSRF framework in the sense that in order for the service to act on it, it will not have to be invoked using a WS-Resource end point reference. In the following subsections it his shown how this cache is populated and managed.

Resource information gathering is among the main tasks of Search Master. The main source of information is the *IS* service, which is accessed through the *IS-Client*. gCube Resources published under the *IS* fall in one of these categories:

- Service
- CS
- Collection
- RunningInstance
- gHN
- GLiteResource

The *SearchMaster* harvests and keeps a local cache of information regarding RunningInstances, Collections and gHNs. Even so, not all of the RunningInstances may be necessary for the *SearchService*. Thus a flag indicating that this Service is of some use to the *SearchService* should be provided by the appropriate services in their profile. This conceptual flag is a *Search Service Profile* XML construct that the Search utilizable services publish, along with their *Service Profiles* or WS-Resource Properties. Furthermore, not all of the information provided in the profile of the above Resource is necessary. For example a service's RunningInstance profile includes the profile of the Service, which includes information on gHN requirements, the service's GAR/WAR and configuration files. For the gHNs, information about current workload is also not needed since the situation may have changed dramatically by the time the node is needed for computation. The *Process Execution Service*, to which the *QueryExecutionPlan* will be handed over, can better tackle such issues. These and other profile fields may prove unnecessary for the *SearchMasterService* needs and may be omitted. The useful information for each of the harvested resources is kept as cache by the *SearchMasterService*. It is expected though that cache freshness issues will arise. The cache will have to be kept up to date. This is a prime issue in the RunningInstances cache which is expected to be subject to change more often than Collections and gHNs.

The local file system of the gHN that each *SearchMasterService* runs on is used as a container for the resources profile Cache. The process of gathering and manipulating the profiles is very similar to the operations employed in a normal search operation. The *IS* service can be considered as a result producer evaluating a query produced by the *SearchMasterService* and delivered through the *IS-Client*. This procedure has to be sensitive to resource updates, removals and new arrivals. Currently this issue is tackled through periodic reharvesting of the existing infrastructure resources, with a period sensitive to cache changes, and with irregular quick checks.

7.3.2.2 Resources and Properties

Although SearchMaster handles information about a large pool of gCube resources, it is a stateless service thus it publishes no IS resources.

7.3.2.3 Functions

As demonstrated in the following diagram the Search Master Service exposes two methods for its clients to interact with.

The main functions supported by the SearchMaster Service are:

- **Search()** – which takes as input parameter a message containing the query (either a Query object serialization or a query expressed using the query language described later in the document) to be executes and returns the ResultSet (its EPR) containing the query results;
- **RetrieveConfiguration()** – which returns an XML document string serialization representing the collections this instance can be used to search over, along with the type or resources that are associated with each collection.

7.3.3 SearchLibrary

The Search Library groups a big set of functional components used throughout the Search family of service. The most important parts grouped are presented below grouped by thematic area, due to the size of the component in discussion.

7.3.3.1 The Query

7.3.3.1.1 Reference Architecture

The *Query* element describes a series of operations to be carried out by the Search engine. The user interface (portlet) and the *QueryParser* are the tools to allow the user to construct a *Query* object by simple means. However, this does not imply that a *Query* object cannot be built by its own. So, there are two methods of automatic and one of manual *Query* object construction. More details concerning the *user interface* and the *QueryParser* can be found at the respective paragraphs.

In general, a query is a form of questioning, in a line of inquiry; a statement of information needs, typically keywords combined with boolean operators and other modifiers; a specification of a result to be calculated from one or more sets of data. In the gCube environment, a query object contains the information for a simple or complex search operation on one or more collections of data.

The principle behind the Query is that for each search operation there is a respective query node class. So, for example, there is a *Join* class which represents the *Join* search operation. Each and every class of the *Query* framework, is a sub-class of the *SearchOperation*. In this way, the system can build query trees, with a query node class define one or more query node classes as its child. The root of this query tree is handled by the *Query* class, which provides methods for (de) serializing a query and locating a specific query node in the tree.

In order to be able to express the appropriate information, a query must describe the following elements:

- Search operation: Every search operation corresponds to a search service, which implements it. Available operations are:
  - FieldedSearch;
  - FullTextSearch;
  - Join;
  - KeepTop;
  - Merge;
  - Project;
  - QueryExternalSource;
  - SimilaritySearch;
  - SpatialSearch;
  - Conditional;
  - Sort;
  - TranformResultSet;
- Source Selection: The Sources Selection part is responsible for identifying the resources on which the Query should be performed defining the collections against which the query criteria should be executed. There are also provisions for the case in

which the submitted query is to be performed against the ResultSet computed by a previous query. Since the ResultSet is to be treated as a WS-Resource, this is done by passing the End Point Reference of the previous result set in the Metadata Source subpart of the Source definition section of any following query.

The *QueryParser* offers a means of construction of a Query object, using a formal, SQL-like language. This is particularly useful for experienced end-users, since it provides much more flexibility and expressiveness than any graphical user interface. However, since it employs a language, it is not addressed to users who do not wish to learn and use that particular language. Apart from the end users, the query parser may be used by any software to automatically build query objects. The QueryParser is responsible for doing the actual parsing, based on a set of rules in BNF format. It is feeded with an instance of the Tokenizer class which holds all of the query tokens. If anything goes wrong during the parsing procedure, then an exception is thrown and thecorrect syntax is displayed in the exception message.

7.3.3.1.2 Functions

All query node classes extend the SearchOperation class. If a class has only one source then it extends the UniSourceSearchOperation class which limits the sources to a single one. If it has more than one sources, then it extends the MultiSourceSearchOperation class which does not poses a lower limit of 2 sources. Last, if a class does not have any sources, then it extends the NonSourceSearchOperation, which simply prohibits all sources. Apart from these methods, each class also exports setters and getters for its own fields. The SearchOperation has only the following method, which is further refined by the Non | Uni | Multi SourceSearchOperation classes:

- **getSources()** – which returns the list of sources of the calling Query node. Actually, it returns a pointer to the SearchOperation List, so that one can add, remove and update it.

The set of operations defined in the Query class are:

- **Query()** – which returns a *Query* object (it is the basic constructor);
- **Query()** – which takes as input parameter an XML representation of a query and returns a *Query* object for it;
- **getRootOperation()** – which returns the root search operation of the *Query* object;
- **setRootOperation()** – which takes as input message a SearchOperation object instance and sets it as the root search operation of the current *Query* object istance;
- **serialize()** – which returns the string serialisation of the current Query object instance.
- **QueryParser()** – which takes as input a string containing the query expression and returns an instance of the QueryParser class representing such a query expression (it is the constructor);
- **parse()** – which returns a Query object instance resulting from the parsing of the query expression the current object has been provided with.

Apart from the functionality of the Library, the main search functionality is accessible throught the operations offered by the Query.

The semantics for each function-operation is the following:

project

Perform projection over a result set on a set of elements.

sort

Perform a sort operation on a result set based on an element of that result set, either ascending (ASC) or descending (DESC).

merge

Concatenate two or more result sets.

join

Join two result sets on a join key, using one of the available modes (`inner`, `leftOuter`, `rightOuter`, `fullOuter`). The semantics of the join modes are the same as in the relational algebra.

fielded search

Keep only those records of a source, which conform to a selection expression. This expression is a relation between a key and a value. The key is an element of the result set and the relation is one of the: `==`, `!=`, `>`, `<=`, `>=`, contains. The 'contains' relation refers to whether a string is a substring of another string. Using this comparison function, one can use the wildcard '*', which means *any*. The source of this operation can be either a result set generated by another search operation or a content source. In the last case, one should use a source string identifier.

full text search

Perform a full text search on a given source based on a set of terms. The full text search source must be a string identifier of the content source.

filter by xpath, xslt, math

Perform a low level xpath or a xslt operation on result set. The math type refers to a mathematical language and is used by advanced users who are acquainted with that language. For more details about the semantics and syntax of that language, please see the documentation for the ResultSetScanner service, which implements this language.

keep top

Keep only a given number of records of a result set.

retrieve metadata

Retrieve ALL metadata associated to the search results of a previous search operation.

read

Read a result set endpoint reference, in order to process it. This operation can be used for further processing the results of a previous search operation.

external search

Perform a search on external (gCube-disabled) source. Currently, Google, RDBMS and the OSIRIS infrastructures can be queried. Depending on the source, the query string can vary. As far as Google is concerned, the query string must conform to the query language of Google. In the case of RDBMS, the query must have the following form, in order to be executed successfully:

```
<root>
   <driverName>your jdbc driver</driverName>
   <connectionString>your jdbc connection
string</connectionString>
   <query>your sql queryt</query>
</root>
```

Finally, in the OSIRIS case, the query string must have the following format:

```
<root>
   <collection>your osiris collection</collection>
   <imageURL>your image URL to be searched for similar
images</imageURL>
  <numberOfResults>the number of results</numberOfResults>
</root>
```

similarity search

Perform a similarity search on a source for a multimedia content (currently, only images). The image URL is defined, along with the source string identifier and pairs of feature and weight.

spatial search

Perform a classic spatial search against a used defined shape (polygon, to be exact) and a spatial relation (contains, crosses, disjoint, equals, inside, intersect, overlaps, touches.

conditional search

Classic If-Then-Else construct. The hypothesis clause involves the (potentially aggregated) value of one or more fields which are part of the result of previous search operation(s). The central predicate involves a comparison of two clauses, which are combinations (with the basic math functions +, -, *, /) of these values

7.3.3.2 QueryPlanner

The objective of query processing in the gCube distributed context is to transform a high-level query into an efficient execution strategy that takes into account the status and nature of the infrastructure, which lies beneath. The need for optimization is evident in many use cases, including the mis-utilization of resources and the beneficial use of existing structures. The task of query processing includes the mapping of the *Query* to an execution plan, consisting of WS invocations, and an initial domain specific optimisation of the produced execution plan. This task is the responsibility of the *QueryPlanner*, which will be thoroughly analysed in this paragraph.

Generally there are two main layers, which are involved in mapping the query into an optimized sequence of local operations, each acting on a specific node. These layers perform the functions of query materialization and query optimization. Before analysing these layers, the most critical part of the query planning procedure, the service profile structure, is introduced.

The most important architectural aspects of the component are presented below

Service Profiles

Each WSRF service that participates in the search framework is accompanied with a profile. This profile contains vital information concerning the usability and applicability of the service. It is primarily used for describing the connection between the service itself and the query it can execute. For example, "*serviceA*" declares in its profile that it can answer to a sort query. During the production of the execution plan from a query tree, the query planner matches the various query nodes to the services which can answer them, thus constructing the execution plan step-by-step.

More analytically, each service, first, describes a set of invocation information, which include its service path name (without the host machine address), its port type and operation and optionally its resource Endpoint Reference (if the service is stateful). Besides the execution information, services declare a semantic descriptor of which queries can they compute. For that reason, XML Schema Definitions (XSD) are employed, so that every service can define the Schema of the XML queries that are able to answer to. Moreover, each service defines a generic transformation of the matching XML query to produce its invocation message (actually the body of the invocation SOAP message). This is done using XSL Transformations of the XML query to the XML SOAP message body. Finally, in order to accommodate the combination of more that one service invocations computing to one sub query, a generic replacement strategy has been introduced, through which a pseudo-service can compute one sub query, by splitting it into many sub-queries, each of which can be directly computed by a single service instance. This is also done using XSL Transformations of the matching XML query to the combination of other XML sub queries.

Service Profiles are basically XML documents that can be found in the DIS component. However, the query planner deserializes them into java classes via a data-binding technology, JAXB. These classes are located in the SearchLibrary. Since the service profiles are only internally used by the Planner component, their design will not be analyzed further.

ExecutionPlan

The output of the Planner component is an instance of the *ExecutionPlan* class, which in principle is just a java graph of service invocations. Edges denote parent-child relationships and nodes specific service invocations. Note that these invocations are abstract in the sense that no specific endpoint reference is defined. To accommodate the WS scheduling that will be performed by the execution engine, the *ExecutionPlan* also includes a set of candidate concrete service endpoint references, so as to be selected later on by the execution engine.

Query Materialization

The main role of this layer is to transform the query operations into concrete search operators, which are provided by the Search Service, escorted by the corresponding gHNs that host these operators. The appropriate information concerning the existent operators and their hosting gHNs is passed to the *QueryPlanner* by the *SearchMasterService*, which in turn receives it from the gCube Information Service. This information is the set of available service profiles, described above, and the set of hosting nodes and their respective services. The generating execution plan is constructed in steps, by matching a query node to the respective search service. There is an M-N relation among sub queries and service invocations, m, n>= 1. The output of this layer is an initial query execution plan which can be forwarded to the Process Execution service (§5.6) and run as is (the BPEL part, see previous paragraph). However, this may lead to poor performance, since no optimization is performed upon the query, which can lead to a substantial execution cost increase.

Query Optimization

The goal of query optimization is to find an execution strategy for the query which is as close to optimal as possible. However, selecting the optimal execution strategy for a query is an NP-hard problem. For complex queries with many sources (collections) involving a large number of operations hosted in a complex infrastructure, this can incur a prohibitive optimization cost. Therefore, the actual objective of the optimization is to find a strategy close to optimal within a logical amount of time. The output of the optimization layer, which is also the final output of the *QueryPlanner*, is an optimized *QueryExecutionPlan* object.

The optimization process followed by the planner makes extensive use of service instances response time statistics and operator cardinalities. Service instances are ranked based on their response times and query operations are ranked based on their selectivity factor. Query Planner enumerates the equivalent plans for a given query and selects the best plan, according to its execution cost. However, an exhaustive search in all possible plans leads to prohibitive time and resource consumption. Therefore, a greedy heuristic of minimizing the cost in each step of the plan construction is excersised. In the first step, the initial query tree is re-written using a set of XSL Transformations, in order to produce a plan with minimum intermediate results size. In the second step, the query tree produced earlier is transformed to its corresponding execution plan, selecting the best service instances that can compute the sub queries of the query tree. If the system comes up with two candidate services that correspond to a given sub query, then it selects only the one that minimizes the total cost up to this point. It has to be noted that only abstract services are selected in this step. The actual scheduling takes place in the third and final optimization step, by the *ProcessOptimizationService* component (*POS*). The *POS* is responsible for allocating tasks to gHNs, based on some criteria, such as gHN load.

To sum up, the *QueryPlanner* gets a query and return a near optimal query execution plan object that contains a sequence of concrete search operations. The final mapping of operations and gHNs will be done at runtime by the execution service described in (*cf. Section* 5.6)

7.3.3.2.1 Resources and Properties

Initially, the Query Planner was designed as a stateless WSRF service. However, due to close relation with the SearchMaster, in the sense that the query planner is invoked for every user query, it is re-designed as a JAVA class embedded in the SearchLibrary. As such no resources are managed by the service

7.3.3.2.2 Functions

The main functions supported by the *QueryPlanner* class is:

- **ProcessQuery()** – which takes as input parameter a Query object instance and returns a PlannerResultHandler resulting from the processing of the Query (which include query decomposition, materialization and optimizati).

7.3.3.3 Execution (Engine) Integration

The term *Execution (Engine) Integration* refers to a logical group of components which materialize the bridge between query planning and plan execution. As previously written, the QueryPlanner produces an *ExecutionPlan* which should be forwarded to the execution engine(s). The procedure of feeding the engine(s) with the *ExecutionPlan* is performed by the components described in this paragraph.

Search-Execution integration is by far not a trivial task because of the obvious fact that components from each side were developed by different groups, resulting in heterogeneity regarding their interfaces. The purpose of decoupling components is exactly the ability to employ them from various institutions. All in all, the problem comes up to the fact that ending up with an *ExecutionPlan* and a set of available execution engines, which potentially have completely different interfaces, one must find a way to feed the engine with a corresponding execution plan, expressed in its native language, but obviously based on the original *ExecutionPlan*. The solution to this integration problem is the introduction of the eXecution Engine Api (XENA), which is described in the following paragraphs. Before that, the available execution engines which are already or will be employed in D4Science are described.

Currently, there are 2 execution engines available in gCube, ProcessExecutionService and the internal QEPExecution engine. The first one is an autonomous service developed separately from the Search Framework. It is thoroughly described in Section 7.3.3.4. The second one is a simple, basic, SOAP engine without any advanced features such as persistence or failover recovery. For further details regarding the QEPExecution can be found below.

Besides these engines, it is important to have the opportunity to employ a widely used, popular, workflow (or better process) engine. ActiveBPEL is selected for this purpose, which is an open-source, BPEL engine, very popular in the web service world. For further details regarding ActiveBPEL, please refer to http://www.activevos.com/community-open-source.php.

7.3.3.3.1 eXecution ENgine API (XENA)

As the name reveals XENA provides the abstractions needed by any engine to execute plans generated by the Search Framework, namely instances of the *ExecutionPlan* class. XENA can be thought of as a middleware between the Search Framework and the execution engine, with the responsibility of 'translating' the *ExecutionPlan* instances to the engine's internal plan representation.

However, the integration issue cannot be treated so simplistically. First of all, one cannot assume that every engine should respect to the XENA paradigm and therefore offer an inherent support/integration to the Search Framework. Another problem is that the

XENA design itself must be very flexible and expressive in order to encompass all of the significant syntactic and semantic attributes of a process execution. Therefore, it is imperative to clarify two major factors: the real-world architectural solution to the search-execution integration and the data model that XENA adopts.

The idea behind XENA is the following: Between The XENA API and the execution engine, there is a proxy component called *Connector*, which is responsible for translating the XENA API model to the engine's internal model. The idea is not new. It has been applied in many middleware solutions and mostly known from the JDBC API. The ala-JDBC paradigm dictates the introduction of connectors that bridge JDBC and the underlying RDBMS. For every different RDBMS product there is a different software connector. So, in the context of search, different XENA connectors for different workflow engines are employed. Each connector, apart from the XENA artifacts, is fed with a set of execution engine endpoint references, in order for the connector to know which engine instance(s) it may refer to. Each XENA connector is first registered in a formal way to a special class called *ExecutionEngineConnectorFactoryBuilder*, publishing itself and the set of supported features. These features are key-value(s) pairs and describe the abilities of the execution engine. They may include persistence, automatic recovery, performance metrics, quality of service, etc. The registration procedure accommodates the dynamic binding to a specific connector (and thus execution engine) made by the *SearchMasterService*, based on some user preferences or predefined system parameteres. So, for example, if a user desires maximum availability in his/her processes, then the SearchMasterService can select an execution engine / respective connector that supports persistence and failsafe capabilities. The dynamic binding of XENA API to a specific connector is accomplished by employing some of the Java Classloader capabilities.

Regarding the data model that is adopted by XENA, it has borrowed the design philosophy of another middleware solution in the area of web service registries, JAXR, a Java Specification (JSR) for that field. The main abstract classes are:

- *ExecutionVariable*: Variables that participate either as data transfer containers or as control flow variables.
- *ExecutionResult*: The current result of the plan execution. The term final is not used, since the ExecutionResult can be retrieved at anytime during the process execution. See the description regarding ExecutioConnector. Through this class, one can get the current ExecutionTrace.
- *ExecutionTrace*: Current trace of execution. It contains the image of the already done/committed actions defined in the execution plan and the ExecutionVariables.
- *ExecutionConnector*: Provides the actual abstraction of the execution engine. It declares some 'execution' methods. The most primitive task is a plain, blocking execute method. However, users may want a more fine-grained control over their process executions. For that reason three Connector levels have been identified:
  - *Basic Level*: it defines a single, blocking execute method, which receives an ExecutionPlan and returns back an ExecutionResult.
  - *Advanced Procedural Level*: Basic Level + process management methods, such as executeAsync, pause, resume, cancel, getStatus, waitCompletion.
  - *Event-Driven Level*: Apart from the purely procedural level, there are many workflow engines which adopt a different paradigm, the event-driven one. According to this, any action that takes place during a process execution, e.g. service invocation finished, or variable initialized, produces a message which can be handled by a corresponding callback method. So the "service invocation finished" event causes the invocation of its associated callback method, within which the developer can perform any management action, housekeeping, logging, etc. The Event-Driven Level offers all the mechanics for registering the set of event which the developer wishes to handle and their associated callback methods. Note that XENA does not offer a callback system. This is the responsibility of the underlying engine. Consequently, only engines that adopt the event model should implement the Event-Driven ExecutionConnector Level.

Since there are three available execution engines (*PES*, *QEPExecution*, *ActiveBPEL*), the initial desing includes three corresponding connectors:

- A connector for the workflow (process) language employed by PES is a shortened version of BPEL.
- An ActiveBPEL connector which fully conforms to the BPEL OASIS standard.
- QEPExecution, internally implemented within the Search Framework and it can directly execute *ExecutionPlans*, (trivial implementation).

7.3.3.4 QEPExecution (Internal Execution Engine)

The QEPExecution is the internal, simple, generic, ws-compliant, execution engine of the Search Infrastructure. It is used in cases of failure or absence of the official *PES* gCube execution engine (or any other BPEL-enabled execution engine).

The *QepExecution* is basically a web service execution engine, which orchestrates the execution of a set of services. It is designed to work with any web service and therefore communicates via the exchange of SOAP messages. The output of the execution is a string of the final results Endpoint Reference (exactly like the output of the *PES*). The *QepExecution* is able to work with any WSRF service (both stateful and stateless). However, since the engine is designed to be used internally as a final solution, it lacks various features of a full fledged execution engine, such as advanced error handling.

The QepExecution engine is employed by the SearchMaster service in cases of failure or absence of the official PES gCube Execution Engine (or any other BPEL-enabled execution engine). The engine is not fed with a BPEL but with the QueryExecutionPlan, which is the raw, internal representation of the execution plan.

7.3.3.4.1 Resources and Properties

Not Applicable

7.3.3.4.2 Functions

The operations exposed by the component are the following:

- **QepExecution()** – which takes as input parameter a message containing an execution plan (and instance of *QueryExecutionPlan* class) and a and returns an instance of the *QepExecution* object;
- **execute()** – which takes as input parameter a boolean value indicating whether to use encryption or not during plan execution and returns a String reporting on the status of the execution.

7.3.3.5 OperatorLibrary

The classes that belong to the SearchOperators set are the core classes of the data retrieval and processing procedure. They implement the necessary processing algorithms and are wrapped by the respective Search Services, in order to expose their functionality to the gCube infrastructure. However, the SearchOperator classes implement only a subset of the whole gCube functionality; there are some services that do not rely on these classes and incorporate the full functionality, without any dependencies to any library (apart from the ResultSet and ws-core, of course). These services are the IndexLookupService, GeoIndexLookupService, ForwardIndex, FeatureIndex and Fuse; they will be analyzed in separate, following paragraphs.

The following diagram depicts the various classes involved in the operator library bundle. Those classes depend on the result set component which is their data transfer mechanism. Finally, the search services classes which wrap around the operator library classes are depicted too.

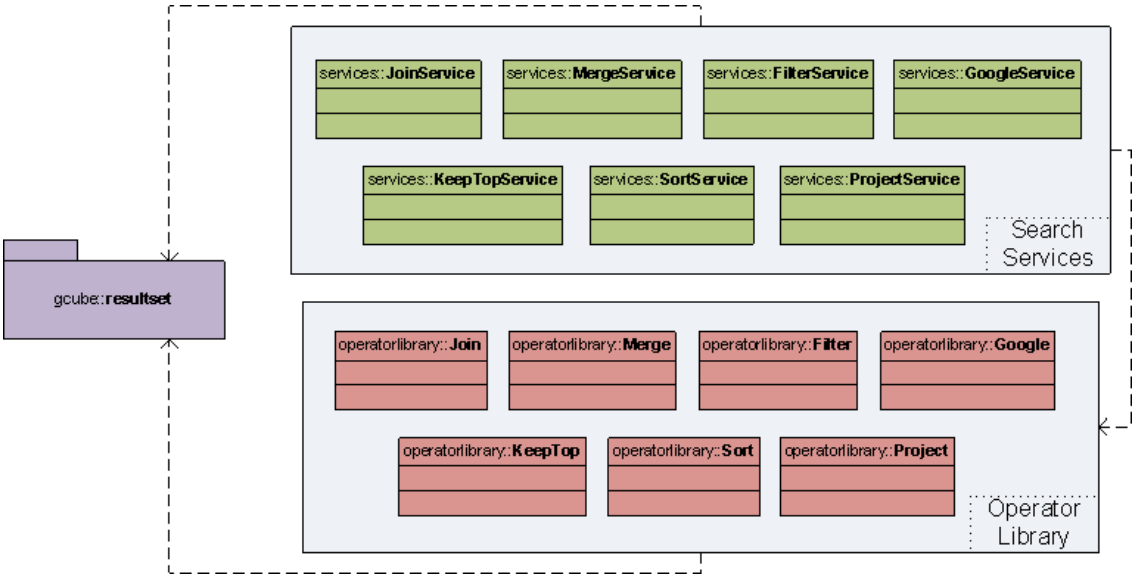

Figure 41. Operator Library Class Diagram

7.3.3.6 QueryPreprocessors

The *QueryPreprocessor* bundle includes the following elements:

- QueryPreprocessorInterface
- CSSQueryPreprocessor
- QueryPersonalizer
- QueryValidator

The above mentioned elements serve the general purpose of performing a series of preprocessing operations on the supplied Query. These include validation, injection of information and appropriate tailoring of the initial Query. As stated earlier in the SearchMasterService, a number of errors and / or warnings may emerge from this procedure as from any other step in the overall Search execution. Depending on the type of error and the context in which it appears, it may not have the same importance and should be handled differently. In the case at hand, an error during the Query preprocessing by the content source selection service, should not stop the entire Search execution, while an error during the Query validation, may have different impact on the Search depending on the type of error.

The *QueryPreprocessorInterface* is meant to be implemented by a number of preprocessors. Each one of these has its own distinct functionality and performs a unique operation on a Query object. However, their overall objective is to refine the context of the search or inject extra information in the query. The refinement includes both accuracy of the query execution results and transformation of the search space in order to either cut off the unnecessary search paths, thus saving time, or add some useful paths, which were not initially selected.

The *QueryPreprocessor* interface declares only one method, the *PreprocessQuery*, which receives a *Query* as input. The various preprocessor elements, which implement this interface will perform a certain operation on this object and return the new *Query*, most probably altered due to the preprocessing procedure. However, in many cases, a preprocessor may not perform any changes on a Query object at all, thus leaving it intact.

Any additional preprocessors that need to be incorporated in the gCube infrastructure, must implement this Interface in order for the *SearchMasterService* to be able to invoke them.

The CSSQueryPreprocessor wrapper covers the functionality of the *ContentSourceSelection* subcomponent of the *DIR* service. This service selects target

collections according to the likelihood that their content will prove relevant to the information need by the underlying query. Estimations of relevance, in particular, are based on summary descriptions of the content of target collections, or collection descriptions. This wrapper is in fact the preprocessor guided by the helper *CollectionSelection* component of the *DIR* service.

QueryPersonalizer wrapper covers the functionality of the *Personalization Service*. It communicates with the *UserProfileAccess* service and retrieves the profile of the user submitting the query. Its responsibility is to process the query in order to bring it close to the user preferences. Currently, the query personalization is limited to affecting the document/result language, field filtering, field ordering and collection(s) addition. The full plan considers to dynamically change user profiles at runtime, based on her query submission history and to add some new functionality to the personalizer component. For further details, refer to the Personalisation (cf. Section 7.5).

The *QueryValidator* element is responsible of validating a specific *Query* object, ensuring that content collections are available, metadata elements are present, operators are accessible, etc. It is provided by the *SearchMaster* with sets of information about the existence and availability of the *Query* parameters and performs the validation procedure against its input *Query* object. If the validation succeeds, then the *Query* will be elaborated by the other preprocessors. Otherwise, the search operation fails in its total.

7.3.3.6.1 Resources and Properties

No resources are created by QueryPreprocessors explicitly.

7.3.3.6.2 Functions

The most important operations exposed the Search Library are the following.

- ***PreprocessQuery()*** returns a new Query object, based on certain operations performed on the Query object, which is passed as input to the method. The implementation of this method is dependent on the implementing element.

As a result of the implementations of the various operatiors the following functionalities are provided by a number of classes that implement QueryOperators

- **BooleanCompare**: implements the hypothesis evaluation class and is wrapped by the BooleanCompareService (cf. Section 7.3.4.2). As previously stated, the hypothesis is performed against an aggregation of one or more field values of the result set of previous queries and may combine these values with some basic mathematical operators (+, -, *, /). The Boolean value is produced by a comparison between combination of aggregated and literal values. For example, one may define "max(fieldA) > 10" as the hypothesis statement.
- **Google**: acts as a wrapper to the Google service, exposing its functionality in a limited way of maximum of 10 results though. The wrapping service is the gCube GoogleService (cf. Section 7.3.4.7).
- **Filter**: implements the xpath result set filtering process and is wrapped by the FilterXPathService (cf. Section 7.3.4.3). Given an xpath expression, this class iterates through the records of a result set and keeps only those that produce a valid xpath result.
- **Join:** implements the join of two result sets process and is wrapped by the JoinService (cf. Section 7.3.4.6). Given a join key this class performs an inner hash join of the records of the two result sets, in a classic relational algebra way.
- **KeepTop**: implements the keep top n process and is wrapped by the KeepTopService (cf. Section 7.3.4.1). This class keeps only as many records as defined in the positive integral argument passed to the class.
- **Merge:** implements the merge process of more than one result sets and is wrapped by the MergeService (cf. Section 7.3.4.5). This class actually performs a concatenation of the various records of the result sets, placing one record after another in the final merged result set.

- **Sort**: Implements the sort process of a result set and is wrapped by the SortService (cf. Section 7.3.4.4). It implements the merge-sort algorithm and is done in memory for efficiency reasons.
- **Transform**: implements the XSL Transformation process of the records of a result set and is wrapped by the TransformByXSLTOperator (cf. Section 7.3.4.8). It iterates through the records of a result set and performs an XLST on each one of them, producing a new transformed result set.

Operators Expose their functionality via as single incvocation:

- ***Compute***: receives the arguments of an operation (probably including EPRs to gRSes) and returns the results.

7.3.4 SearchOperators

The various intermediate steps towards producing the final search output are handled by *SearchOperator* services. These include Fielded Search operators, FullText Search operators, ForwardIndex Search operators, FeatureIndex Search Operators, result Sorting operators and others that will be analyzed in the following section. All these operators are implemented as stateless services. They receive their input and produce their *ResultSet* output in the context of a single invocation without holding any intermediate state. This is not necessarily the case with the Index, FeatureExtraction, Feature Index and DIR-DataFusion Services which are detailed in later sections.

The *SearchOperators* cover the basic functionality that could be encountered in a typical search operation. A search can be decomposed in undividable units consisting of the above operators and their interaction can construct a workflow producing the net result delivered to the requestor. The external source search and the service invocation services provide some extendibility for future operators by offering a method for invoking an "unknown" to the Search Framework, importing its results to the search operator workflow. The distinguished search operators at present time are listed below.

7.3.4.1 KeepTopService

The role of the *KeepTopService* is to perform a simple filtering operation on its input *ResultSet* and to produce as output a new *ResultSet* that holds a defined number of leading records. This operation is accomplished through the use of the respective *SearchLibrary* operators package facility.

7.3.4.1.1 Resources and Properties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.1.2 Functions

The main functions supported by the component are:

- **KeepTop()** – which takes as input parameter a message containing a ResultSet and an integer indicating the number of desired records and returns a new ResultSet (its EPR) containing the count top records of the input ResultSet.

7.3.4.2 BooleanCompareService

The BooleanCompareService is used in conditional execution and more specifically, in evaluating the condition. So, it actually offers the ability of selecting alternative execution plans. For example, one can follow a plan (let's say a projection on a given field of a set of data), if a given precondition is valid; otherwise, she may follow the alternative plan (e.g. a projection on another field of the same set of data and then sort on the field). The precondition validation is the responsibility of the BooleanCompareService.

The condition is a Boolean expression. Basically, it involves comparisons using the operations: *equal*, *not_equal*, *greater_than*, *lower_than*, *greater_equal*, *lower_equal*. The comparing parts are either literals (date, string, integer, double literals are

supported) or aggregate functions on the results of a search service execution. These aggregate functions include *max*, *min*, *average*, *size*, *sum* and they can be applied to a given field of the result set of a search service execution, by referring to that field employing xpath expressions.

This operation is accomplished through the use of the respective *SearchLibrary* operators package facility.

7.3.4.2.1 Resources and Properties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.2.2 Functions

The main functions supported by the *BooleanCompareService* are:

- **compareMe()** – which takes as input parameter a message containing a set of ResultSet (their EPR) and a control condition (this expression/condition defines the comparison, the aggregate functions, the resultset EPRs and the xpaths) and returns true or false, depending on whether the condition is valid or not.

7.3.4.3 FilterXPathService

The role of the *FilterXPathService* is to perform search through an expression to be evaluated against an XML structure. Such an expression could be an *xPath* query. The XML structure against which the expression is to be evaluated is a *ResultSet*, previously constructed by an other operator or complete search execution. The result of the operation is a new *ResultSet* and the end point reference (EPR) to this is returned to the caller. This operation is accomplished through the use of the respective *SearchLibrary* operators package facility.

7.3.4.3.1 Resources and Properties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.3.2 Functions

The main functions supoported by the FilterXPathService component are:

- **ExecuteQuery()** – which takes as input parameter a message containing a ResultSet (its EPR) and an XPath expression to be evaluated and returns a ResultSet containing the records resulting from the evaluation of the provided input.

7.3.4.4 SortService

The role of the *SortService* is to sort the provided *ResultSet* using as key a specific field. This operation produces a new *ResultSet* leaving the input untouched. The newly created *ResultSet* is wrapped around a WS-Resource and its end point reference is returned to the caller. The algorithm used is merge sort. The comparison rules differ depending on the type of the elements to be sorted. This operation is accomplished through the use of the respective *SearchLibrary* operators package facility.

7.3.4.4.1 Resources and Properties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.4.2 Functions

The main functions supported by the SortService component are:

- **SortResultSet()** – which takes as input a message containing a ResultSet (its EPR), the field whose values has to be used as key and the criterion (Ascending or Descending) and returns a new ResultSet whose records are sorted accoding to the values of the specified field and criterion.

7.3.4.5 MergeService

The role of the *MergeService* is to perform a merge operation using a set of *ResultSets* whose end point references (EPRs) are provided. This operation produces a new *ResultSet* leaving the input untouched. The newly created *ResultSet* is wrapped around a WS-Resource and its endpoint reference is returned to the caller. This operation is accomplished through the use of the respective *SearchLibrary* operators package facility.

7.3.4.5.1 Resources and Properties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.5.2 Functions

The main functions supported by the MergeService component are:

- **MergeResultSet()** – which takes as input parameneter a message containing a set of *ResultSet* (their EPRs) and returns a new ResultSet (its EPR) resulting from the union of the input *ResultSets*.

7.3.4.6 JoinService

The role of the *JoinService* is to perform an inner join operation on a specific field using a set of *ResultSets* whose end point references (EPRs) are provided. This operation produces a new *ResultSet,* leaving the input untouched. The newly created *ResultSet* is wrapped around a WS-Resource and its endpoint reference is returned to the caller. An in memory hash – join algorithm has been implemented and is used through the respective SearchLibrary operator facility.

7.3.4.6.1 Resources and Properties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.6.2 Functions

The main functions supported by the JoinService component are:

- **join()** – which takes as input parameter a message containing a set of ResultSet (their EPRs) and a field whose values are to used to identify the matching records and returns a new ResultSet (its EPR) resulting from this conditional union activity.

7.3.4.7 GoogleService

GoogleService acts as a simple wrapper around the Google® functionality. It receives a user query (plain string) and forwards it to the Google service that does all the actual work. Unfortunately, due to legal constraints, a maximum number of 10 results can be retrieved.

7.3.4.7.1 Resources and Porperties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.7.2 Functions

The main functions supported by the GoogleService are:

- **search()** – which takes as input parameter a message containing the query to be evaluated by the Google search engine and returns a new ResultSet containing the records returned by Google (the top 10 only) as the result of the specified query.

7.3.4.8 TransformByXSLTOperator

The role of the *TransformByXSLTOperator* is to transform a *ResultSet* it receives as input from one schema to another through a transformation technology such as XSL / XSLT. These transformations are directly supplied as input to the service. The output of the transformation, which could be a projection of the initial *ResultSet*, is a new *ResultSet* wrapped in a WS-Resource whose endpoint reference is returned to the caller. This

operation is accomplished through the use of the respective *SearchLibrary* operators package facility.

7.3.4.8.1 Resources and Properties

This service produces a volatile ResultSet (i.e. not registered in IS).

7.3.4.8.2 Functions

The main functions supported by the TransformByXSLTOperator are:

- **ProjectXSLT()** – which takes as input parameter a message containing a ResultSet (its EPR) and a transformation specification (i.e. directives to transform the records constituting the input ResultSet into a different format from their originator one) and returns a new ResultSet (its EPR) containing the records transformed in the target format.

7.3.4.9 IndexedLookup

The role of the *IndexedLookup* is to perform a search against a forward index of a collection. The operation is handled by the *Lookup* operation defined in the Index Management section (cf. Section 7.4). The Index Lookup will return an endpoint reference to the *ResultSet* it produces, wrapped as a WS-Resource, which is passed back to the caller. Information on this service is provided in following sections.

7.3.4.10 GeospatialSearch

This is a specialized case of Index Search and its role is to perform a search against an index of a spatial collection capturing overlapping of geographical areas. The operation is handled by the Lookup operation defined in the Index Management section (cf. Section 7.4). The Index Lookup will return an endpoint reference to the *ResultSet* it produces, wrapped as a WS-Resource, which is passed back to the caller. Information on this service is provided in following sections.

7.3.4.11 FullTextSearch

This is another specialized case of Index Search and its role is to perform a search for content by using special inverted indices. The operation is handled by the *Lookup* operation defined in the Index Management section (cf. Section 7.4). The Index Lookup will return an endpoint reference to the *ResultSet* it produced, wrapped as a WS-Resource, which is passed back to the caller. Information on this service is provided in following sections.

7.3.4.12 FeatureExtraction

The role of the FeatureExtraction is to invoke feature extractors for on-line or batch / on-demand extraction. Existing feature extractors are available for image, audio, video, etc. The output, if there is any, is wrapped around a *ResultSet* and exposed as a WS-Resource, whose endpoint reference is returned back to the caller. Information on this service is provided in following sections.

7.3.4.13 FeatureSearch

This is a specialized case of Index Search and its role is to perform a content based information retrieval operation on feature vectors extracted by the *FeatureExtractionService.* The operation is handled by the Lookup operation of the *SimilarityIndex* Service. The Index Lookup will return an endpoint reference to the *ResultSet* it produced, wrapped as a WS-Resource, which is passed back to the caller. Information on this service is provided in following sections.

7.3.4.14 DataFusion

The role of the DataFusion is to fuse the ranked ResultSets passed to it as input and produced by a Full Text Index. The output of the DIR/DataFusion Service is a *ResultSet,*

whose end point reference is returned back to the caller. Information on this service is provided in following sections (cf. Section 7.6).

7.3.5 Search Deployment Diagram

As far as deployment of search components is concerned, each Search Operator Service is dependent at runtime from the operator library package of the Search Library and the Result Set Service. All Result Set Service instances communicate in pairs in order to perform the data transfer. Finally, the Search Master depends on the Search Library and the Result Set Service.

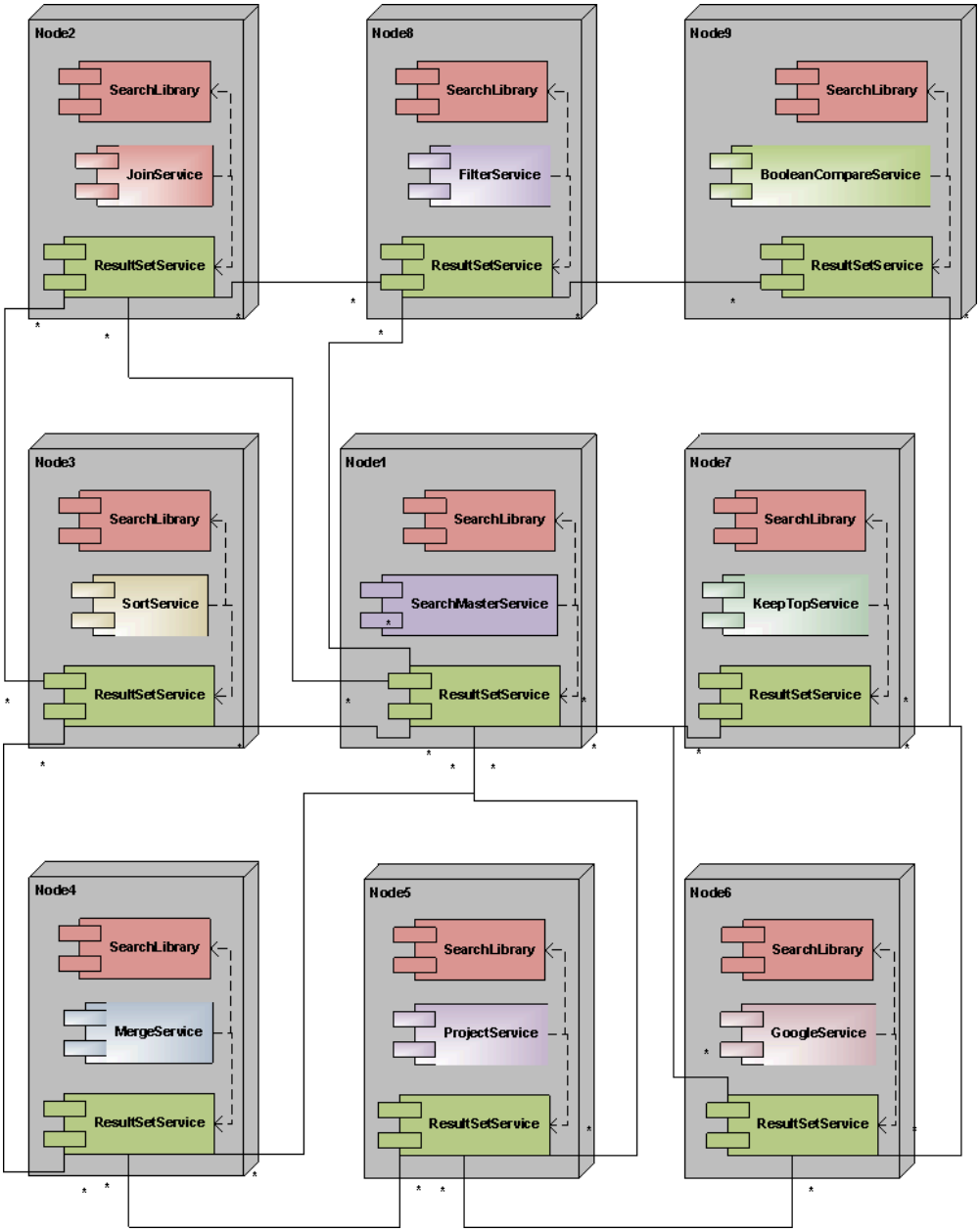

Figure 42. Search Deployment Diagram

7.4 Index Management Framework

7.4.1 Reference Architecture

The Index Service is the entity in charge of building and maintaining full-text indexes of content. In addition to full-text indexes, forward indexes are supported according to the requirements of the project as well as indexes to facilitate geo-spatial search. Finally high-dimentional indices, appropriate for efficient handling of feature-index vectors, are implemented for supporting content-based search over images.

As full-text indexes are metadata of collections, the index service collaborates with the metadata management service to associate indexes with collections. The overall concept of an index is simple: indexes are used to accelerate certain types of access to a collection of digital objects. Lookup operations on an index take as input a query, compliant with the query language of the particular index, issued by the search service and output a ResultSet, i.e a sequence of occurrences that satisfy the predicate and are typically ordered according to a ranking function supported by the index and displayed in a way parameterized by the query. Secondarily, the index is capable of returning aggregated statistics of the index occurrences to enable federation of the result sets from distinct indexes by the data fusion service.

The design of the index service intends to preserve this abstract simplicity, and yet cover the considerable range of variability needed for concrete indexes to support the feature and performance requirements. It is expected to incorporate legacy index implementations, which partially may need to be statically installed (managing their own storage), whereas other implementations will fully support the infrastructure by utilising the underlying storage and management layers.

The Index Service is broken down into packages that contain gCube services as shown in the following figure. Actually each package contains gCube services. The factory – instance pattern is used for creating/managing resources in the gCube services.

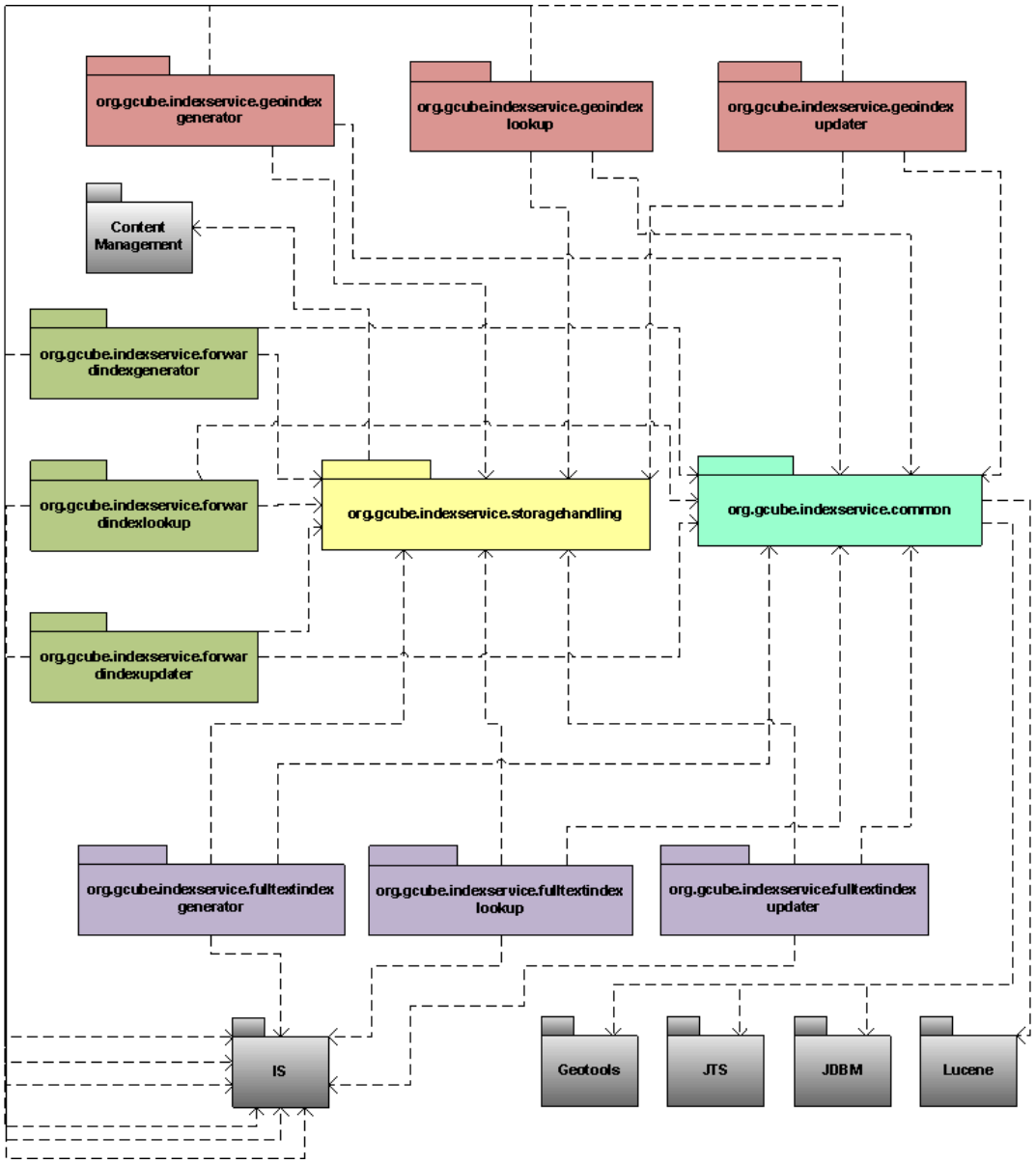

Figure 43. Index Service Logical view

7.4.1.1 Design Considerations

Targeting extensibility and openness as design paradigms, the design accommodates new index functionality and new implementations of indexes, making neutrality and stability of interfaces key design objectives. In practice, this means that whenever protocol elements depend on index variations, these variations are described in extendable XML schemas, and do not affect the overall signatures of the service interfaces. Typical index variations are:

- Types of objects being indexed;
- Document model: Types, names and semantics of searchable fields;
- Query language;
- Ranking function.

The full text Index Service has been designed to rely on the information retrieval library Lucene, which is an Open Source Software project from Apache Jakarta. The Lucene library is not a search engine itself, but it provides functionality for implementing search

engines. The Lucene library is rich in features and extendable for handling full-text search.

The geospatial index service has been designed to rely on the Geotools library, which provides standards-compliant methods for the manipulation of geospatial data, based on R-tree structures. Geotools is managed as an open source project, and delivers a Java code library with a LGPL license. It implements the Open Geospatial Consortium specifications. Furthmore, the geospatial index utilizes the JTS Topology Suite, an open source API that provides 2D spatial algorithms. It delivers a Java code library with a LGPL license.

The forward index service has been designed to use JDBM, a transactional persistence engine for large object collections, which provides scalable data structures such as B+Trees. It is fast, simple, robust with a small footprint and is therefore considered to be suitable to be used on the Grid.

Finally the feature index service has been implemented natively in gCube in order to provide a repository for Feature Extraction products and an engine for fast lookup in similarity searches. The solution that powers the index is a variation of the VA File construct, which is proven to offer better performance than other multidimentional indices in the case of similarity search.

These external libraries are wrapped by the Index Service in order to make them suitable for running on the gCube architecture.

Following the aforementioned extensibility and openness considerations, the interfaces to the Index Service are designed in such a way that the Index Service gains maximum flexibility.

7.4.1.2 Index Types

The Index Type defines the index structure and manages what features the index provides. This process is similar to defining a schema or table structure for a database.

The index type specifies each field that is contained in the index and, for each field, a list of attributes that describe what kind of information is stored. Fields in a feed that are not described in the corresponding index type will not be indexed (will not be present in the index), and thus it will not be possible to search these fields. The field attributes are described in the table below.

Table 1. Index Type Field Attributes

Property	***Description***	***Possible values***	***Default***
Name	The name of the field	Lowercase letters and numbers. First character must be a letter	
Type	The type of the field	string, uint32, int32, float	string
Stored	Determines the form of the original content of the field	static, dynamic, no	static
Indexed	Whether or not the field is indexed	yes, no	no
Tokenized	Whether or not the field is tokenize	yes, no	no
Sort	Specifies whether search result can be sorted by this value	yes, no	no

Boost	Value that sets the field boost factor used in the ranking formula	Float	1.0

7.4.1.3 The RowSet document format

The Index Service expects to receive documents that are to be indexed in a RowSet format. The URI of the RowSet files is sent as part of SOAP messages to the Index Service.

The RowSet element encapsulates individual documents, each of which is encapsulated in ROW xml elements. For each ROW element each field described in the specified Index Type are listed. The field elements are text elements which contain the content of the various fields. In some cases it is feasible to reference the content with an URI; in this case the attribute type="URI" in the field element is used and the URI where the content is available is provided as the text value.

7.4.1.4 Delta Files

The index is maintained as a collection of delta index files, meaning that the index is divided into smaller parts; the delta index files are stored by relying on the facilities of the gCube Information Organisation Services (cf. Section 6). The reasons for introducing the concept of delta files in the index design are:

- The partition of the index is supported through the use of delta files;
- It is more convenient to store and manage a large index as a number of smaller files in order to avoid size limitations in the content management component as well as in Grid file transfer utilities;
- It is more convenient to transfer a number of smaller files instead of a large file when the index is transferred over the network from safe storage in the content management to a storage node, especially if the file is transferred over an unreliable network;
- In the case that a storage node has limited storage capacity, the use of smaller delta files makes it more flexible to partition the index on several nodes instead of one node.

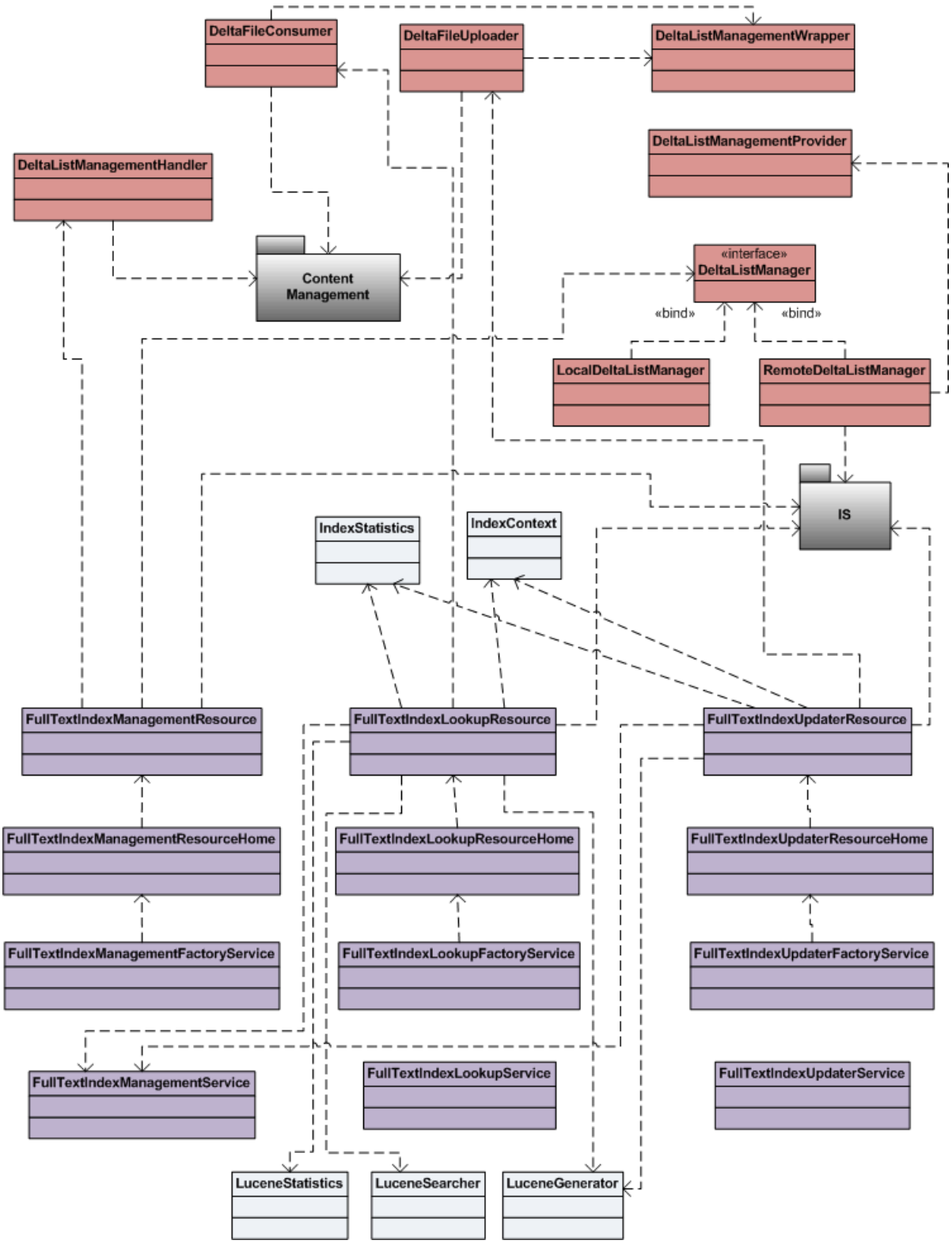

Figure 44. Full Text Index Services Class Diagram

The design allows for adding implementations built on top of other information retrieval systems. Another information retrieval library would be implemented by registering another name for the corresponding factory web service in the Index Type Manager, adding data (Index Type and Index Format) describing when and how to use the new

implementation and installing the corresponding gar/jar containers in the gCube Information System.

7.4.1.5 Deployment View

The Index Service contains three components:

- The Generator
- The Updater
- The Lookup

The architecture supports a distribution where each component may be distributed on the same node or all components on different nodes. These distributions are trivial and therefore not shown here. They may be applicable in a test/lab environment.

In a real environment there are two different (orthogonal) ways to scale a search engine:

- Scale with the size of the index (the number of documents in the index). This setup needs a component distribution where the VRE contains one Generator and several Update components and implies that the index is partitioned over several nodes. The partitioning of the index puts requirements on the document feeding process (content management) that has to keep the overview over partitions and the documents that each partition contain.
- Scale with the number of queries per seconds. This setup needs a component distribution where the VRE contains several Lookup components and implies that each index partition is replicated to several nodes.

Experience shows that to support a large index, the index must be partitioned over several nodes and in addition each partition must be replicated to several nodes (i.e. the above methodologies are combined).

This is shown in the following Figure, where one set of nodes is responsible for updating a partitioned index and another set of nodes are responsible for index lookup. The index is copied from the node generating and updating the index to each node in the column (using for example the features in Content / Storage Management). In addition to the components of the Index Service, that are shown in the deployment view, the Index Service is dependent on a number of components like the Information System, the Content/Storage Management, and components in the Search Service like the Search Library and the ResultSet Service.

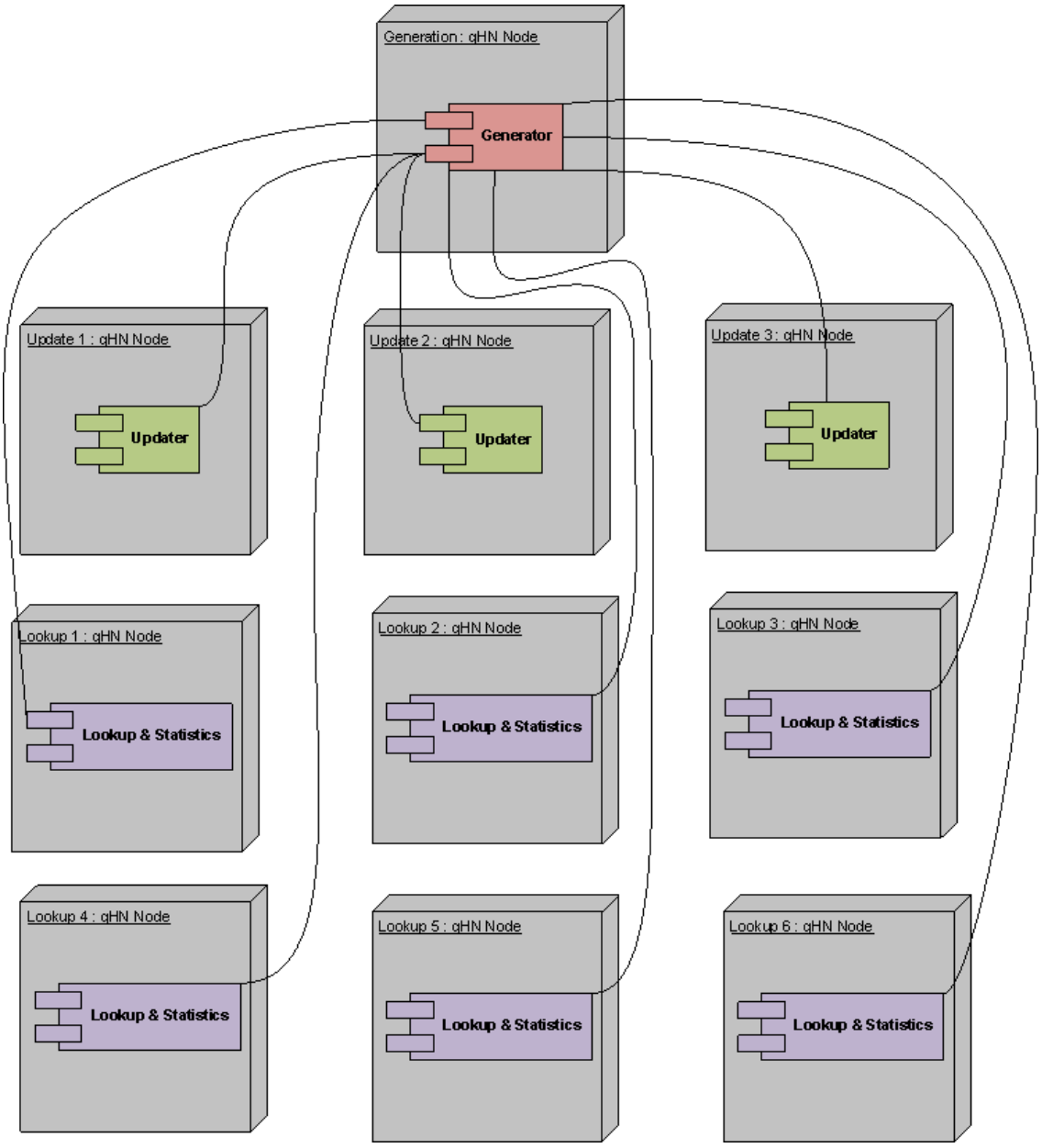

Figure 45. Index Service Deployment View (Large Scale Deployment)

7.4.2 Index Common Library

The Index Common library exposes functionality used by the other index components in order to carry out common tasks. There are several utility classes which simplify the handling of statistics, thread management and index type management. This library also contains wrappers for every external dependency used by the various indexes, such as Lucene, JDBM and GeoTools, thus exposing an interface for communication between index components and external libraries.

7.4.2.1.1 Resources and Properties

Not Applicable.

7.4.2.1.2 Functions

The functions supported by index Common Library are to a large degree exported directly by the services that host it and can be found below.

7.4.3 Index Storage Handling Layer (Library)

The Index Storage Handling component is responsible for partitioning the index data in delta files, storing them in the content management layer and retrieving them when required. All the index services are built on this core component, so that data partitioning is performed transparently for every index. The storage handling component also offers notification handling and brokering fanctionality, thus allowing other index components to be notified whenever there is a change in the data of an index.

7.4.3.1.1Resources and Properties

Not Applicable.

7.4.3.1.2 Functions

The main functions supported by the library are wrapping around the CMS information access functions (i.e. read and write operations).

7.4.4 Index Generator

The index subsystem is composed of three different components, namely the full text index, the geospatial index and the forward index. Each one of these components is designed using the same pattern and contains three services: the index generator service, the index update service and the index lookup service.. The index generator service contains the logic to generate an index.

7.4.4.1.1Resources and Properties

It maintains one resource for each index. This instance is in charge of the index, and keeps the list of delta index files. The instance is created when the index is created, and is deleted when the index is deleted.

7.4.4.1.2 Functions

The main functions supported by the service are:

- **createResource()**: intializes the creation of an index resource
- **getIndexType()**: gets the Index Type of the index.
- **SetIndexType()**: set the index type of the index
- **getIndexInformation()**: retrieves important characteristics for the index
- **addCollections()**: adds a String id to the collectionID resource property, essentially creating the link among the index and a collection

7.4.5 Index Update

The updater service is responsible for updating an existing index. Index updates are performed in cooperation with the generator service instance that originally created the index. The update operation is supported both as an incremental updater and a batch updater. The incremental update operation depends on notifications identifying the changed content. The batch update operation accepts operations to modify the content. Only one entity can update an index at any time. Therefore, the execution of the operations (that may run as separate web services) must be coordinated and synchronized before running the update operation against the information retrieval library.

7.4.5.1.1 Resources and Properties

The updater service maintains one resource for each collection that feeds content into the index. Several resources may exist in parallel feeding data into the same index. The Batch Updater resource receives the documents in the Row Set format, and creates a

delta index file that represents a part of the index. When the feeding process is committed, or the file size grows above a certain size, the file is stored in content management, using the Storage Handling. If the feeding continues a new delta index file is opened, and the procedure of storing the delta index file is repeated.

7.4.5.1.2Functions

The main functions supported by the service are:

- **createResource():** creates a WS reference endpoint of the type Updater service.
- **processResultSet():** Adds one or several documents to the index using the rowSet format.
- **Process():** Process Row Sets as described above, but the Row Sets are received as a parameter in the Process operation.
- **Commit():** Finishes the delta index file and forces a merge of the file index into the main index. The content in the main index is available for querying.
- **Abort():** Discards the temporary index, destroys the index BatchUpdater and release temporary resources. The delta index file is not stored in CMS, and will not be a part of the index.
- **DeleteDocument():** Delete an entry from the index.
- **FinishUpdate():** the sequence of process/processRow Set operations is complete. The documents can be merged and stored.

7.4.6 Index Lookup

The lookup service offers data retrieval functionalities over existing indexes. The output of the operation is passed to the client in the response. The index is not changed by operations in this package (read only), and therefore it is possible to instantiate several WS reference endpoints of this type on the same computing node working on the same index. The index lookup service operates on a local copy of the index being created by replication through the content management service. The package also provides simple global statistics for an index.

7.4.6.1.1Resources and Properties

The factory creates new lookup resources for an index on request. The create request contains a reference to the service that is in charge of the index. The lookup resource is deleted when the index is deleted, or it can be manually deleted by the operator.

The Index Lookup WS resources register for notifications and receive a notification when the list of delta index files is modified. The Lookup service gets the new/modified file from CMS using the Storage Handling, and merges the delta index files into the index.

7.4.6.1.2 Functions

The main functions supported by the service are:

- **createResource():** This operation creates a WS reference endpoint of the type full Text Index LookupService
- **Query()**:Looks up the query in the index and the ranked results and per result statistics in the ResultSet.
- **createStatistics():** Create the overall statistics for the index
- **getIndexType():** Gets the index type.
- **setIndexType():** Sets the index type in the resource.
- **expandIndex():** Expands the index with a new partition.

7.5 Personalisation Service

The term "Personalisation" refers to the ability of a system to adapt its behaviour to the personal needs and preferences of its users, i.e. expose a different behaviour for each and every user that accesses it.

The topics being addressed by the personalisation layer in a VRE Management System[11] may include:

- Personalisation of the information retrieval behaviour;
- Personalisation of user-system interaction (through the presentation layer);
- Personal storage provision.

The most widely used technique for supporting such operations is the adoption of user profiles, which includes the following two topics:

- Population and retrieval of user profiles
  - Automated (intelligent)
  - Manual
- Management of profiles (creation, deletes, transformations etc)

The gCube personalisation service provides the appropriate means for persisting user profiles in the storage management system. It focuses on the Information Retrieval aspects of the topic, by designing and providing the framework to build highly sophisticated personalisable Virtual Research Environments.

Personalisation in gCube is handled by using user profiles, i.e. "records" that contain the information necessary for adapting the system's behaviour on a per-user basis. These profiles are created and maintained by explicit system and user operations. Although creation and maintenance of profiles for gCube is 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 available to various services for their own use. The Portal Engine and the portlets it hosts exploit the user profile as a container for persisting aspects of the presentation configuration. Such configuration includes the configuration of the front-page portlets, the layout of each portlet which can be different depending on user's selections, the language, the default collections where user searches are submitted, the user preferences for resource utilisation, etc.

Personalisation of information retrieval has two stages:

- Declaration stage: where the rules to guide the retrieval are stated;
- Retrieval stage: where the rules are being applied.

The rules are stated as elements in the user profile, in particular to the section dedicated to QueryPersonalisation. Such information includes the content sources to be utilised, sorting preferences, language preferences, etc.

During the retrieval stage, there are various options for applying the aforementioned rules:

- Pre-selection of sources;
- Transformation of output to specific languages;
- Pre-presentation processing of information;
- Implicit filtering of information;
- Influencing of Search Service internal behaviour.

When a new user profile is created, it contains a number of mandatory elements, which are defined in the default user profile. The default user profile is stored in the gCube-

[11] VRE Management System is referred to as an analogy to an (R)DBMS that handles the resources, managed the data and offers all the constructs that end-user or developers will need to access the data/ information hosted.

powered infrastructure as a generic resource. However, due to its large scope of application, the user profile can be enhanced with many other elements.

7.5.1 Reference Architecture

The personalisation service is divided into two services, the ***UserProfileAccess*** service and the ***ProfileAdministration*** service.

For a better understanding of the structure of the personalisation service (*i*) an operational diagram, which provides a high level representation of the service, and (*ii*) a class diagram, which provides a more in-depth representation of the service, are given below.

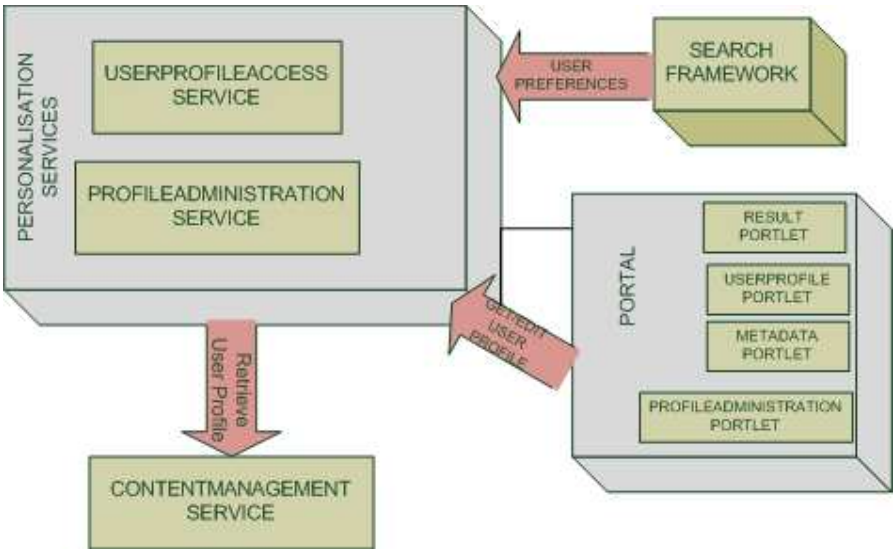

Figure 46. Personalisation Service Operational Diagram

Both services invoke the content management service for retrieving the user profile which is stored as a file in the gCube's storage system. When the profile is created it can be accessed by different portlets and/or by gCube services in order to retrieve the user's preferences and adopt the system's behaviour. The access of the profile is available by invoking the appropriate personalization service. Each service provides different functionality.

The following diagram depicts the internal structure of the services.

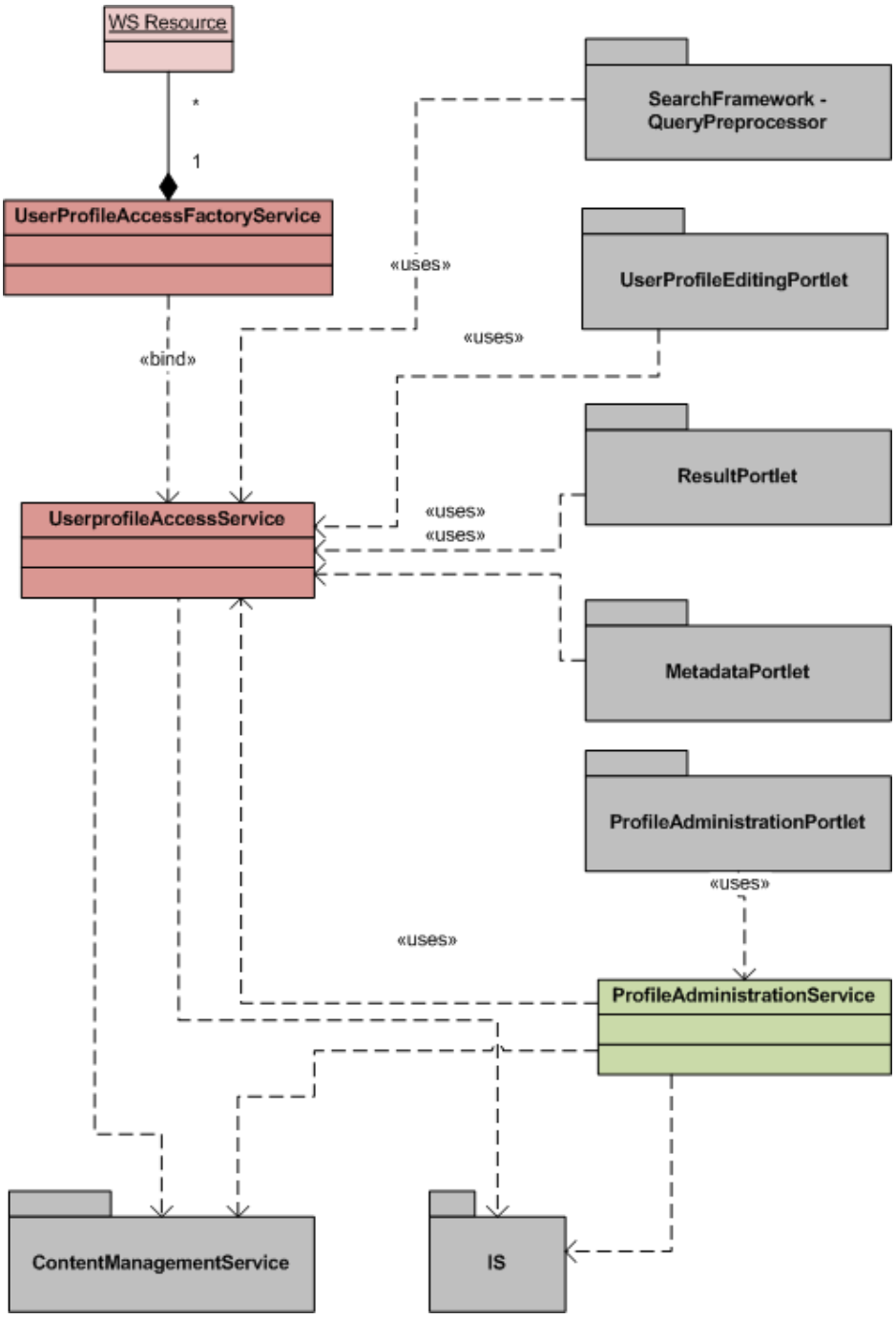

Figure 47. Personalisation Service Class Diagram

The deployment of these two services is independent. They can be deployed in the same or in a different node. Considering the distribution of the system, it is proposed that these two services will be deployed in different nodes.There are no other deployment dependencies. The content management service, which both services depend on, can be deployed in another node.

7.5.2 UserProfileAccess Service

The *UserProfileAccess* service is responsible for creating and managing the user profiles, so all the non-administrative components are invoking this service.

7.5.2.1 Resources and Properties

UserProfileAccess service is a stateful service. It follows the factory pattern for web services and has two components, the factory service which is responsible for creating WS-Resources, one for each user profile, and publishes them on the IS. Each WS-Resource has two properties, the user's username which is used as the logical filename of the profile and it is stored in the Storage Management System and the ID of the user profile, which is the identifier returned by the Content Management Service when the profile is stored in the Content Management Layer. The second component provides all the functionality for retrieving and updating a user's profile.

In order to create a new user profile the factory component must be invoked to create a new WS-Resource. Then the userProfileAccess component can access and change the profile by retrieving the appropriate WS-Resource from the IS which gives direct access to the profile.

7.5.2.2 Functions

The main functions provided by the UserProfileAccess Service are:

- **getUserProfile** – which returns the user's profile. The user is known because this method is invoked using the WS- resource that had been created already.
- getElementValue – which takes as input the name of an element of the user profile and returns the value of this element.
- **getElement** – which takes as input the name of an element of the user profile and returns an array which contains all the elements having the input's name.
- **setElement** – which creates a new element in the user profile under the specified path, depending on the user's input.
- **setElementValue** – which takes as input the name of an existing element of the user profile and the new value to be set.
- **deleteElement** – which takes as input the name of an element of the user profile and deletes this element.

In order to create a new user profile the user should invoke the above method from the UserProfileAccessFactory service

- **createResource** – which takes as input the username of the new user profile and creates a new user profile returning the address of the WS-resource

7.5.3 ProfileAdministration Service

The *ProfileAdministration* service allows the administrator to create and/or delete a user's profile and to update the default user profile. This service is invoked by the profile administration portlet only.

7.5.3.1 Resources and Properties

The ProfileAdministration service which is an administrative service, invokes the UserProfileAccess service in order to create a new user profile. For deleting an existing profile it can directly access the WS-Resource and delete it. This will also delete the profile from the Storage Management system. The last functionality of this component is the creation/update of the default user profile, which is stored as a generic resource on the IS. In order to provide this functionality it communicates with the IS service.

7.5.3.2 Functions

The main functions provided by the ProfileAdministration Service are:

- **createUserProfile** – which takes as input a username and creates a new user profile with this username.
- **dropUserProfile** – which takes as input a username and deletes the profile with this username.
- **setDefaultProfile** – which sets or updates (if already exists) the default user profile which is stored as a generic resource on the IS.

7.6 Distributed Information Retrieval Support Framework

Support for Distributed Information Retrieval (DIR) subsumes functionalities that seek to increase the efficiency and the effectiveness with which unstructured queries are evauated across a number of independent collections.

Given a set of such queries and two or more target collections for their evaluation, there are three main functionalities that fall squarely within the scope of DIR support:

- **collection selection** is the identification of the target collections which appear to be the most promising candidates for query evaluation. The process may be defined in relation to *goodness criteria* and *selection criteria*: the first are used to rank the target collections from the most promising to the least promising, while the second are used to select collections based on their rank. Depending on the choice of goodness and selection criteria, the process may be geared towards improving the efficiency or else the effectiveness of query evaluation. In the first case, the output serves to limit the costs of query evaluation only to the most promising collections. In the second, the output is used to regulate the number of results retrieved from each collection, and thus to limit the number of irrelevant results which may be presented to the user. The two usage models may well coexist in a single collection selection strategy.
- **result fusion** is the integration of the partial results obtained by evaluating the queries against (selections of) the target collections. Typically, the process is one of harmonisation of result scores that have been assigned with respect to different retrieval models and content statistics. As such, the process is geared towards the effectiveness of query evaluation.
- **collection description** is simply the synthesis and maintenance over time of summary information about the content of target collections, from partial inverted indices, to language models, to result traces for training or past queries. The process is only of indirect interest in the context of query evaluation. Its output forms the basis upon which collections are ranked before being selected for query evaluation. It *may* also be required to normalise the scores with which the partial results of query evaluation are finally merged.

7.6.1 Reference Architecture

The **DIR Master** service embodies the simplest approach to provide functionality of collection selection, collection description, and result fusion within a gCube infrastructure. The service participates of the runtime search framework in two different roles (cf. Section 7.3): as a query pre-processor, it is invoked to perform collection selection prior to query evaluation; as a search operator, it is invoked to merge results after the query has been evaluated across selected target collections. Within the implementation of the service, collection selection and result fusion are implemented by local algorithms. The algorithms, however, rely on the availability of collection descriptions gathered from other Information Retrieval services asynchronously with respect to the query evaluation workflow. In particular, Index Management services and Content Management services are relied upon to localise descriptions of the target collections. The relationships between the DIR Master service and other Information Retrieval services are illustrated in the component diagram of Figure 48.

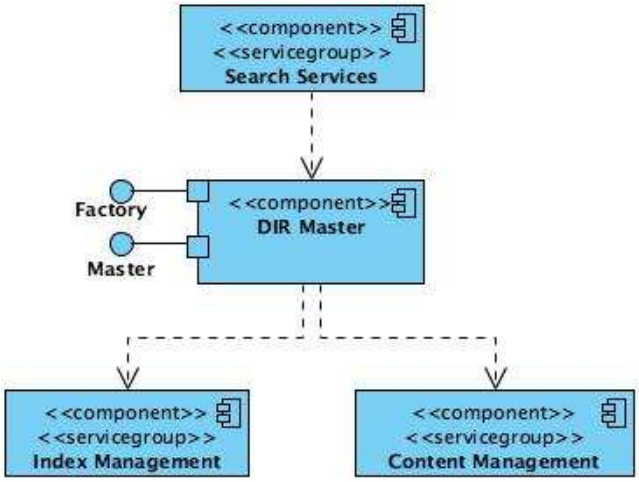

Figure 48. DIR Master Service

7.6.2 The DIR Master Service

The design of the service is distributed across two port-types: the Master and the Factory port-types.

The **Master** port-type has the primary task of applying collection selection and result fusion algorithms to a set of target collections. Locally, such sets are materialised in terms of collection 'proxies' – containers of summary information about the remote collections – and are maintained both in memory and on the file system. Collectively, they form the state of the port-type and are bound on a per-request basis to it, in line with the implied resource pattern of WSRF. In particular, the pairing of the Master interface and the collection sets identifies WS-Resources referred to as *Masters*.

Masters are created in response to client requests to the **Factory** port-type and inherit their scope. It is within this scope that a selection of their properties is published in the Information System (cf. Section 5.2). In WSRF terminology, these are the Resource Properties that identify: (i) the collections in the bound set, and (ii) the selection and fusion algorithms supported by Masters. At creation time or at any later point in the Master's lifetime, clients may add or remove target collections from the associated set.

Adding a collection to the Master requires interactions with WS-Resources and Running Instances of Index Management and Collection Management services. The interactions are entirely based on instantiations of the Handler's framework included in the gCF, and thus inherit best-effort and caching strategies from it. A successful interaction results in the local availability of a *term histogram* of the collection, i.e. a dictionary of the stems of the most content-bearing words of the collection content, each annotated with their frequency of occurrence within the collection. The histogram is then ingested in the *master index*, a local inverted index of the union of the content of all the target associated with Masters.

From the master index, collection content statics can then be retrieved for the application of selection and fusion strategies. These are identified dynamically on a per-request basis. For example, a collection selection strategy begins with the application of an algorithm to rank the target collections with respect to a client-specified query. This is then followed by the application of a selection criterion, also specified by clients, against the resulting collection ranking. Differently from selection criteria, however, ranking algorithms are identified implicitly, by comparing characteristics of the individual request (e.g. the type of the query and query terms) with those of prototypical examples of the inputs expected by the available algorithms. This dynamic approach allows the port-type to present a very extensible interface and thus: (i) support multiple algorithms at any one time, and (ii) evolve to support more algorithms whilst maintaining backwards compatibility. Similar considerations apply to merging strategies.

In the current version of the service, Masters rank target collections by estimating the likelihood that their content will prove relevant to the information needs that underlies queries. The *collection retrieval inference network* (CORI) is a collection-level generalisation of the Bayesian Inference Network probabilistic model of retrieval for text documents. In the model, probabilities are based upon statistics that are analogous to term occurrence frequency (*tf*) and inverse document frequency (*idf*) in classic document retrieval. In particular, term frequency is replaced with *document frequency* (*df*) and inverse document frequency is replaced with *inverse collection frequency* (*icf*). Three selection criteria may then be chosen to return the first *n* target collections in the ranking. In the *TopN* criterion, *n* is an absolute constant. In the *BestScores* criterion, *n* is derived with respect to a threshold on the relevance score. In the *ResultDistribution* criterion, *n* is derived from a distribution of the number of results to be returned to the user.

As to fusion, the current version of the service guarantees a *consistent* merging of the results sets that emanate from the target collections in response to the evaluation of a given query. In particular, Masters *re-compute* the relevance estimates of the documents identified in the result sets using non-heuristic techniques. To do so, Masters rely on: (i) global collection content statistics available in the master index, and (ii) term occurrence statistics for each result (such as the number of terms in the corresponding document and the frequency with which the query terms occur in the document). Effectively, the availability of collection-wide and result-wide statistical information allows the service to consistently re-rank the virtual collection comprised of all the documents identified in the result set.

8 THE PRESENTATION SERVICES HIGH-LEVEL DESIGN

8.1 Overall Architecture

The overall architecture of the presentation services is based on the Application Support Layer (ASL), which provides full functionality for discovering, accessing and interacting with all gCube services. The ASL is an intermediate layer and lies between the presentation applications and the gCube services.

The ASL provides an Application Programming Interface (API) that can be used for developing a new presentation application (i.e desktop or web application) using the presented framework. This is feasible because the applications are not aware of the underlying infrastructure. The ASL handles the invocations to the underlying services and provides functionality in terms of simple java methods and http calls. The upper layer is responsible only for the presentation layer.

For a better understanding of the architecture described above, an operational diagram is given below.

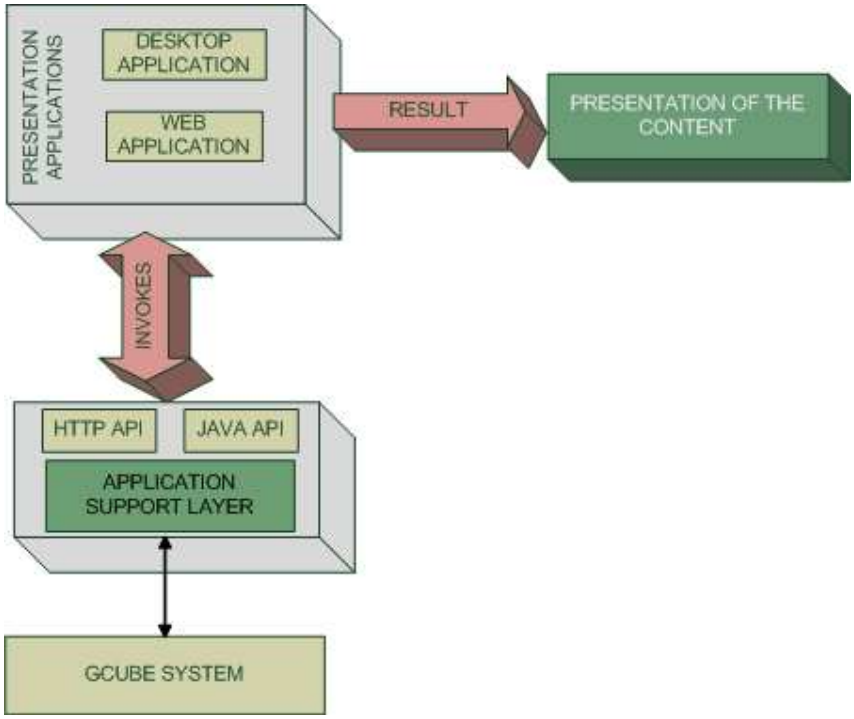

Figure 49. Presentation Layer Operational Diagram

As it is shown in the above figure, the applications that are created using the ASL communicate only with this layer, and they don't know about the lower levels. They are only responsible for the presentation of the content that is fetched through the ASL.

8.2 Application Support Layer

The Application Support Layer (ASL) is a middleware between gCube Lower level Services and the gCube Presentation Layer. Its main goal is to provide a unified interface for accessing gCube and to allow application developers to deal only with how to render and present the corresponding functionality. Furthermore, ASL is responsible for covering all the application logic needed in order to fulfill users' requirements. Since ASL aims at addressing a wide audience - not only gCube developers, it is designed to wrap gCube's complexity by packing and interfacing standard procedures like retrieving

service running instances from IS, attaching (or not) caller credential to stubs, and in general, to prepare the environment for a remote call. It should be mentioned that ASL consumers need no knowledge about where the services are running, whether a remote call is required in order to accomplish their goal, etc.

Offering both Java and HTTP APIs, it supports multiple ways to contact and interact with gCube lower level services. In essence, ASL acts as a gateway for other technologies to access gCube facilities. This means that it receives all the requests to call gCube services and, acting as a proxy, it invokes the corresponding services.

8.2.1 Reference Architecture

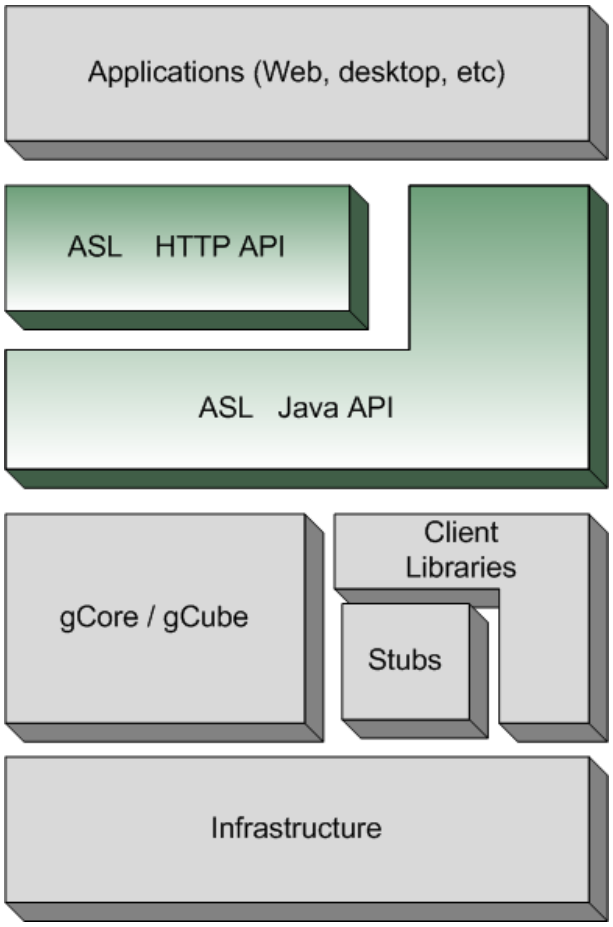

Figure 50. Application Support Layer Placement In gCube Architecture

In brief, the design of ASL is motivated by the following objectives:

- To remove the application logic from the presentation layer.
- To enable external developers to create Graphical User Interfaces (GUIs) for gCube system without any knowledge of its internal design and architecture.
- To speed up gCube system's performance by caching frequently used, and/or rarely modified resources.
- To provide an easy way to manage credentials.
- To offer a global session management that enforces interoperability and intercommunication between application components.

The aforementioned objectives and design decisions are analyzed in the following sub-sections.

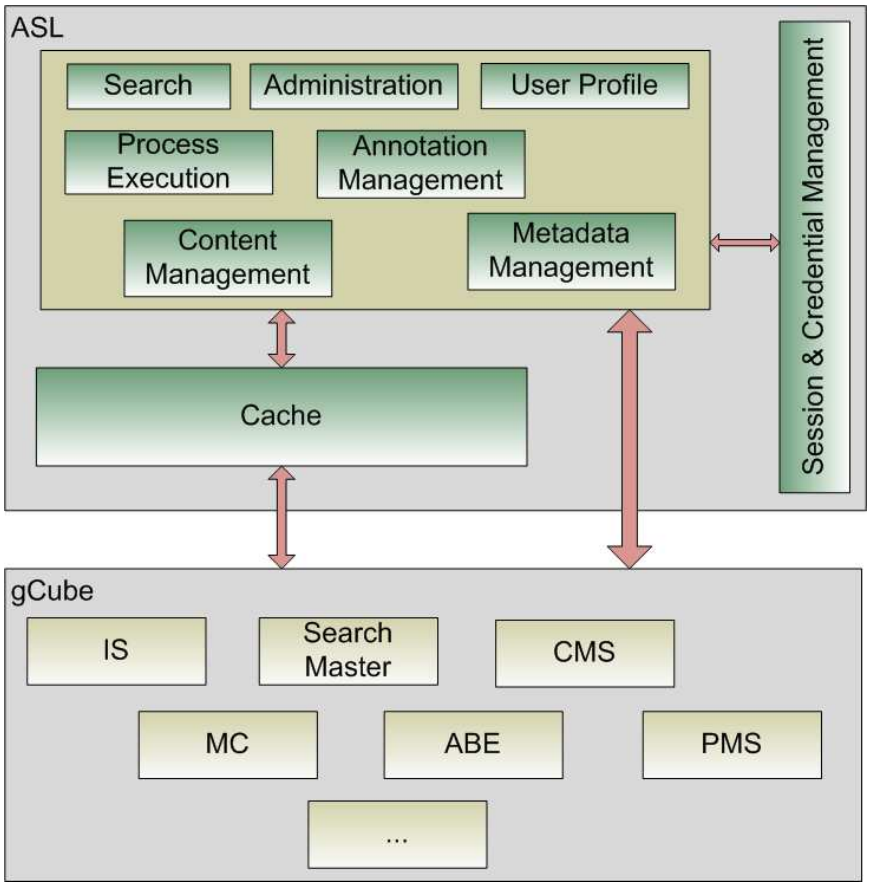

Figure 51. ASL Architecture

As it is depicted in Figure 51, ASL is internally organized in three main groups: session and credential management, cache, and the core functionality.

8.2.1.1 Session and Credential Management

An important part of ASL is the session and credential management. In order to consider ASL as a framework that completely supports user's needs, it should implement its own mechanisms for providing sessions and credentials.

Session is of crucial importance for the presentation layer since it provides means of storing information related to each user and means of intercommunication between application components. For this reason, a **Session Manager** has been implemented, based on the *singleton* design pattern. Its role is to administrate session objects, and to provide each user with their session. By using it, an application component can intercommunicate with another component by storing information in the session, and reading it from the other one. In fact, session is a hash map containing name-value pairs.

Another significant aspect of ASL is credentials and their maintenance. The gCube system is based on PKI credentials in order to authenticate and authorize users. As a result, creating, retrieving and safely keeping proxy certificates is a central goal. ASL transparently retrieves user's proxy credential and stores it in the session. Additionally, depending on the security type of the active VRE[12] (secure or not), the ASL decides whether or not to attach user's credential on the service's stubs in order to make a secure call. In summary, ASL consumers don't need to have any knowledge about how

[12] Active VRE is defined as the VRE on which the user is currently working.

security is implemented in gCube, how to retrieve credentials, make secure calls to services, etc.

8.2.1.2 Cacheing for Performance

Cache plays a main role within ASL. Its existence is considered essential in order to improve gCube's performance and to give the impression to the end-user that the application layer is always ready to promptly respond to their requests. Therefore, a variety of resources that are frequently requested are cached in order to speed up response time. Depending on the type of the cached resource, different refresh and storage policies can be applied. In particular, one can configure the expiration time of the cached resources, the maximum number of these objects in memory, whether or not to support overflow on disk, the maximum size of objects on the disk, etc. Furthermore, examples of resources that is planned to be cached are:

- *The available collections per VRE*, ASL caches the names of the available collections in a hierarchical structure together with information about the available corresponding metadata, and the existence of any special kind of index, like full text, geospatial, or similarity index.
- *Content information*, it contains information about the digital objects, like their mime type, length, the existence of annotations over the object, the number of associated metadata, etc.
- *Generic Resources*, it caches generic resources from IS since they are frequently requested and yet rarely modified. Examples of such resources are the metadata presentation xslts, the record presentation xslts, etc.
- *Metadata*, the metadata are stored in cache so as to be retrieved quickly. In many cases, the user requests to see the metadata of a digital object in a specific schema and then he/she requests this object's metadata in another schema. So, if the alternative metadata exist in cache, the system responses immediately.
- *Profiles*, user's profile is used in many cases in order to personalize the layout based on user's preferences. Thus, having a replica in main memory (or in the local disk) speeds up the procedure.
- *Searchable fields per metadata schema*, for each metadata schema the VRE administrator can define a set of fields that the user can search. Typical examples of such fields are the title, the description, and the author.
- *Thumbnails*, they are typically used for presentation reasons when result records are presented. They are stored as alternative representations in Content Management System, and as a result, their retrieval is a big overhead. Therefore, keeping a copy locally improves application's performance.

8.2.1.3 Core functionality

The core functionality is composed of a set of subcomponents that pack together related functionality. Particularly, it consists of Search, Metadata, Content, Annotation, Process Execution, User Profile and Administrative functionality groups. In reality, it is the API that ASL consumers use in order to interact with gCube. It acts as a proxy that hides complexity of several steps needed for the invocation. It also contains the application logic for interpreting information received by gCube and transforming it into a presentable form.

Detailed description of core functionality can be found in the next paragraph.

8.2.2 Functions

As it is already mentioned, ASL provides both Java and HTTP APIs. The Java API lies on top of gCube, and of any available client-side libraries that offer functionality over gCube services. On the other hand, HTTP API lies only on top of ASL's Java API. In reality, it is an alternative way to provide the same functionality. The rationale of such an architectural decision is that in many cases building an interface over the most well

known internet protocol provides clients (end-users) with an easy way to remotely consume resources.

In the following subparagraphs, detailed information regarding Java and HTTP APIs can be found.

8.2.2.1 Java API

Java API wraps all the gCube functionality needed by the presentation layer so as to fulfill users' requirements. The complete set of the elements that compose the API can be identified on the next diagram.

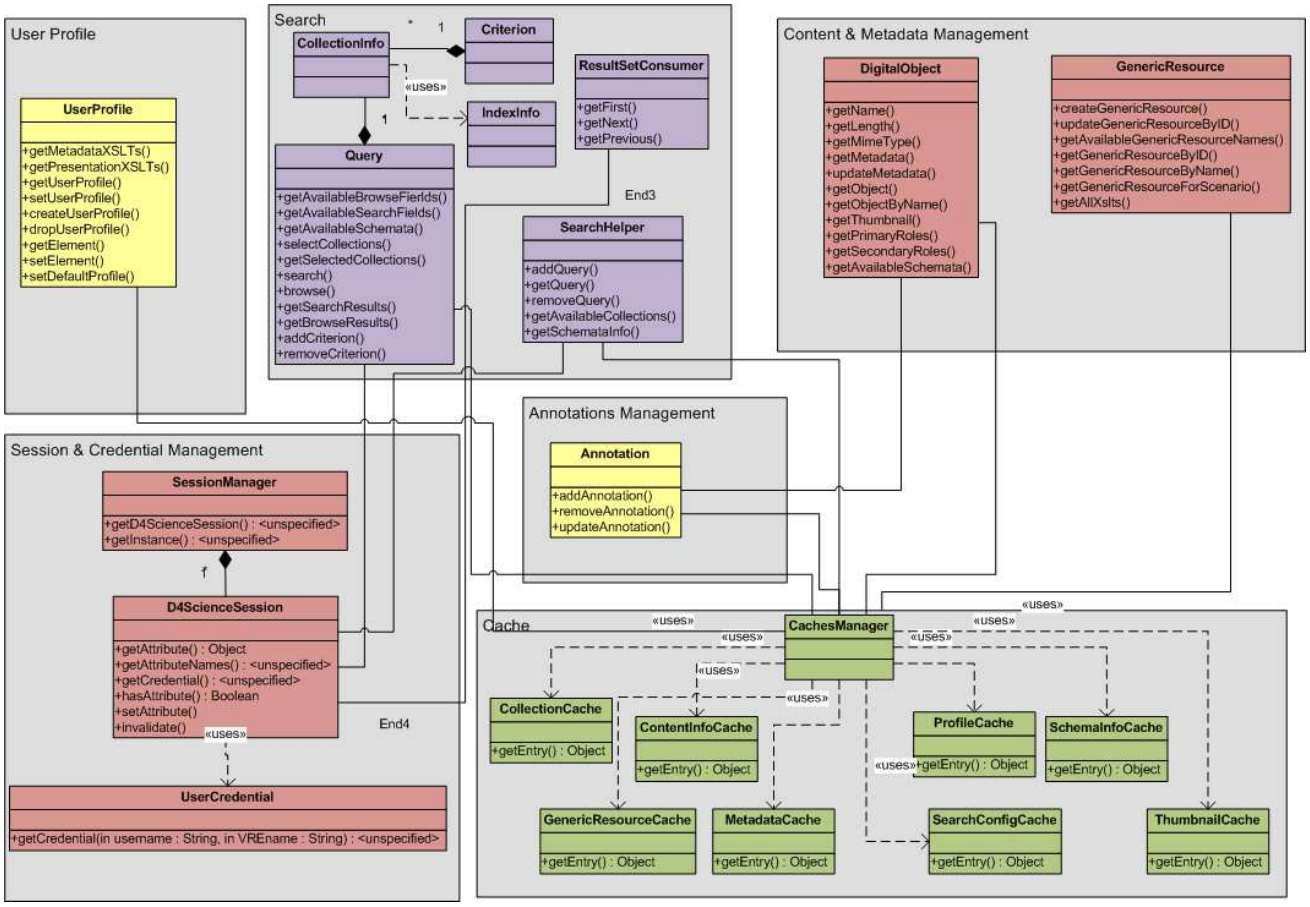

Figure 52. ASL Class Diagram

A thematically grouped short description of selected functionality follows in the next .

8.2.2.1.1 Search

The design of search support took into consideration that multiple queries must be able to be edited, and as a result, multiple result-sets must be available. Thus, each query is constructed on a different object, and all of them are stored in the session.

- **getAvailableCollections():** returns the available collections for the corresponding VRE.
- **getQuery():** returns a specific query object.
- **getAvailableSearchFields():** returns the available search fields for this query depending on the selected collections.
- **setSelectedSchema():** sets the selected metadata schema which will be used to query the colletions.
- **addCriterion():** adds a new criterion to the query.

- **search():** submits the query to *SearchMaster*.
- **browse():** retrieves all the objects of the requested collections.
- **getFirstResults(n):** returns the first 'n' results of the resultset.
- **getNextResults(n) :** returns the next 'n' results of the resultset.
- **getPreviousResults(n) :** returns the previous 'n' results of the resultset.

8.2.2.1.2 Content Management

- **getObject()**: retrieves the digital object's content.
- **getThumbnail()**: retrieves a thumbnail for the corresponding digital object based on the requested resolution. Thumbnails are stored in the Content Management System as alternative representations of the object itself..

8.2.2.1.3 Metadata Management

- **getMetadata():** returns the metadata associated with a specific digital object.
- **updateMetadata():**upd**a**tes the existing metadata of a specific digital object.

8.2.2.1.4 Annotation Management

ASL reuses the existing annotation front-end (AFE) library in order to support the fundamental annotation functionality.

- **getAnnotations():** provides the annotations of a specific digital object.
- **removeAnnotation():** deletes an annotation.
- **addAnnotation():** adds a new annotation and associates it with a specific digital object.
- **updateAnnotation():** updates the content of an existing annotation.

8.2.2.1.5 User Profile Management

- **getElement():** gets the value of the defined element.
- **setElement():**sets the value of the defined element.
- **getMEtadataXSLTs():**retrieves the xslts used to produce an html representation of the metadata.
- **getPresentationSXLTs():**retrieves the xslts used to transform each result record to html.
- **setDefaultUserProfile():**sets the default user profile.

8.2.2.1.6 Administrative Functionality

Contains all the functionality needed to build GUIs for administrative use.

- **queryIS():** supports a mechanism for submitting queries to IS and retrieving the results.
- **deployDL():** deploys the DL.
- **getCollection():** gets the DL model of the corresponding Digital Library.
- **getMetadataRelatedToCollection():** gets the metadata collections related to content collections that will participate in the corresponding DL.
- **setCollection():** sets the DL model of the corresponding Digital Library.
- **setMetadataRelatedToCollection():** sets the metadata collections related to content collections that will participate in the corresponding DL.
- **store():** stores a software package to the software repository.
- **set():** retrieves a software package from the software repository.
- **approve():** approves a software package to be part of the software repository.

8.2.2.2 HTTP API

HTTP API offers a subset of Java API, mostly the fundamental functionality needed by simple end-users. It is planned to be extended based on users' needs and requirements.

It consists of a set of servlets that supports both GET and POST http methods depending on what functionality is requested.

In detail, it currently provides support for the following functionality:

8.2.2.2.1 Search

- ***Query Submission:*** The servlet receives all the needed components in order to construct the query; it creates the query object and submits it by using the java API offered from ASL.
- ***Results Retrieval:*** The results are returned either in xml form, or in html depending on the query parameters. Additionally, since the volume of the result records can be arbitrarily big, only a subset of the results is returned each time. The user can define which subset they want to retrieve in the http query parameters.

8.2.2.2.2 Content Management

- ***Content Retrieval:*** retrieves the content of the corresponding digital object.
- ***Thumbnail Retrieval:*** retrieves a thumbnail for the corresponding digital object based on the requested resolution.

8.3 The gCube Portal Engine

To provide the end user with the full functionality of the gCube system, a presentation application, based on the ASL only instead of the gCube services directly, has been implemented.

For this application the portal/portlets paradigm has been adopted. 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. The portlets are the visual components that participate into providing the user-conceived functionality of the portal. Behind the portal there is always an engine (and a framework) that powers the system.

In gCube we use the GridSphere portal framework [26] is employed as the portlet-hosting platform. GridSphere is an open-source framework which 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 JSR 168 Portlet API, to provide reusable web applications.

For the development of portlets several technologies are involved: the JSR 168 portlet API, Java Server Pages for dynamically generation of HTML/XML documents in response to a client's request and GWT (Google Web Toolkit) for writing high performance AJAX applications. All these technologies are hosted under the Gridsphere Portal engine.

8.3.1 Portal Technologies

The development of a presentation application does not require using any specific technology, but each developer can adopt his/her preferred technologies. In our implementation of the portal framework as it has been already mentioned, the JSR 168 (which is hosted under the Gridsphere portal engine) is used and also the JSP and GWT technologies. A description for these technologies is given below.

8.3.1.1 JSR 168

JSR (Java Specification Request) 168 establishes a standard API for creating portlets. It constitutes the integration component between applications and portals, and it enables delivery of an application through a portal. Without this standard, each version of an application needed its own portlet API, and each of the various portals required that these portlets should be specifically tailor-made for that portal.

8.3.1.2 Java Server Pages (JSP)

Java Server Pages (JSP) is a Java technology that allows developers to dynamically generate HTML, XML or other types of documents in response to a Web client request. The technology allows Java code and certain pre-defined actions to be embedded into static content.

The JSP syntax adds additional XML-like tags, called JSP actions, to be used to invoke built-in functionality. Additionally, the technology allows for the creation of JSP tag libraries that act as extensions to the standard HTML or XML tags. Tag libraries provide a platform independent way of extending the capabilities of a Web server.

JSPs are compiled into Java Servlets by a JSP compiler. A JSP compiler may generate a servlet in Java code that is then compiled by the Java compiler, or it may generate byte code for the servlet directly. JSPs can also be interpreted on-the-fly reducing the time taken to reload changes.

8.3.1.3 Google Web Toolkit (GWT)

Google Web Toolkit (GWT) is an open source Java software development framework that allows web developers to create Ajax applications in Java. It is licensed under the Apache License version 2.0.

GWT emphasizes reusable, efficient solutions to recurring Ajax challenges, namely asynchronous remote procedure calls, history management, bookmarking, and cross-browser portability. For all these reasons most portlets use the GWT for a better and more efficient Ajax implementation.

8.3.2 Portal Deployment

The deployment of the portal has some dependencies that need to be deployed in the same node.

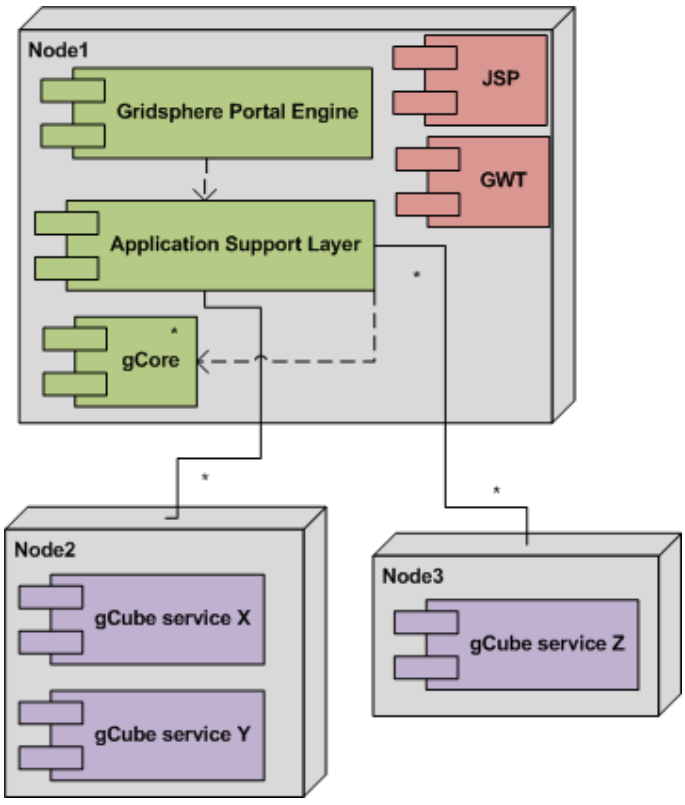

Figure 53. Portal's Deployment Diagram

The implementation of the presentation layer requires a node where one has to deploy some mandatory components. The gridsphere portal engine which hosts all the developed portlets, need to be deployed together with the Application Support Layer and all the technologies that portlets use in a gCore-enabled node.

The ASL communicates with various gCube services, but none of them has to be deployed in the same node.

It has to be noted that this diagram refers to the case where the communication between the portal and the ASL is done through Java calls. In a different case, where a presentation application uses the HTTP protocol to communicate with the ASL, the above deployment diagram does not state this case. The developed application can be deployed on its own node and the ASL on a different gCore-enabled node. The only components that might have to be deployed in the same node with the application, is the technologies that it uses for the development.

8.4 gCube Portlets

The developed portal application consists of various portlets. Each of the portlet follows the pattern described above. The presentation layer belongs to the portlet and the logic layer is handled by the ASL.

Different portlets provide different functionality and all of them together are providing a full functional Virtual Reseacrh Environment, where users can perform a search, retrieve the results, refine the results, browse the available collections, view the metadata of the collections, annotate them and create reports. These features can be also parameterized by using the personalization portlets to declare their preferences on various aspects.

In the following sections each portlet is described separately. Furthermore, the various portlets are presented in Appendix B.

8.4.1 Search Portlets

Search portlets aim to provide a graphical user interface to enable users to search the digital objects. Depending on the results that users want to retrieve there are different search portlets available to use. The Collection navigator portlet is responsible for providing the user with a GUI so as to select the desired collections for applying the search. Therefore, it visualizes the collections in a hierarchical structure helping the user to identify collections with similar content. The Simple search portlet provides an easy way to query the gCube system allowing the user to submit a full text search on the metadata of the selected collections without spending any extra effort. The Browse portlet provides the browse operation on the selected collection. The Combined search portlet provides a way to submit complex queries which are composed of different field criteria either merged or joined together.

After a search is performed, the user can see the results using the Browse Results Portlet. Then he/she can use the Previous Result Search portlet to refine these results. This portlet provides a way to query an existing set of previous results and receive more specific results.

8.4.2 Quick and Google Search Portlet

The quick search portlet provides the ability to the end user to search through the gCube system. This portlet differentiate from the other search portles because the search is not applied on selected collections, but on all available collections of the VRE. The google search portlet provides a way to search google directly from the gCube portal.

8.4.3 Generic Search Portlet

The generic search portlet is a portlet that refers to more technical and advanced users. To perform a search using this portlet the user should be aware of the syntax of the gCube's query language (cf. Appendix A), the IDs of the content and also the available

indexes. Then he/she can perform a query and retrieve the results from the Browse Results Portlet.

8.4.4 Browse Results Portlet

This portlet aims to provide common navigational functionality through the results. In addition, a set of actions can be performed on each result record, like accessing metadata, add annotations, saving or viewing the content, etc.

The user can navigate throughout the results by using the links: *first*, *previous* and *next*. These links correspondingly get the first, previous or next 10 results.

For each result, one can see a set of information, retrieved from object's metadata. The record's presentation form is user configurable. One can select the preferred layout for results, by using the User Profile Editing portlet. Additionally, a thumbnail (small representation of the content) appears at the right part of each result record.

Moreover, one can perform a set of actions on each result:

- ***View Content***: It opens a popup window that contains, and displays the content;
- ***Save Content***: Forces browser to ask to save the content in your local disk;
- ***Manage Annotations***: add/remove/edit annotations for this object;
- ***View Metadata***: view record's metadata;
- ***Edit Metadata***: edit record's metadata.

8.4.5 User Profile Editing Portlet

The user profile editing portlet provides a graphical interface which allows users to view and/or edit their profiles. Using this portlet a user can change his/her profile depending on his/her preferences. He/she can change tha way the results/the metadata are being displayed, he/she can define search restrictions (i.e the language of the collections he/she wants to search in).

Personal information for each user are also available through this portlet.

8.4.6 Profile Administration Portlet

The Profile Administration portlet can be accessed only by users who have administrative rights. Using this portlet, the administrator can create and/or delete a user profile by typing the user's username. The Administrator is also responsible by uploading the Default User Profile on the IS as a generic resource. This can be done through this portlet. The Default User Profile should be a valid XML document.

8.4.7 Metadata Broker Administration Portlet

The Metadata Broker Administration portlet is a graphical user interface that provides a way to manage transformation programs. Its main functionality is to create, edit and delete transformation programs. The list of available transformation programs is retrieved from the IS.

When the user creates a new transformation program or edits an existing one, the portlet is in “edit mode”, allowing the user to specify the transformation program’s name, its description as well as its full XML definition.

8.4.8 Annotation Front End

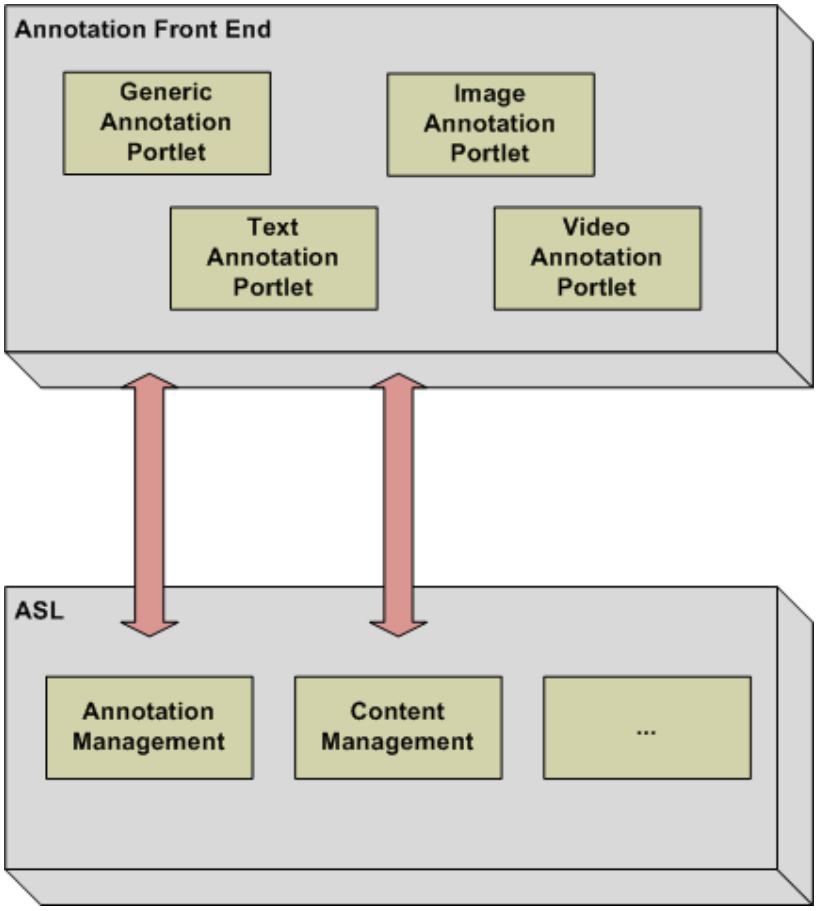

Figure 54: Annotation Front-End Architecture

Annotation Front End is a graphical user interface that provides an easy and sufficient way to annotate digital objects. Its main functionalities are view, add, edit and delete annotations of an object specified by the user.

Annotations can be anchored and non-anchored meaning that a user can annotate only a part of the object for some particular file types (anchored annotations) or the whole object for every file type (non-anchored annotations).

Annotation Front End consists of the Generic Annotation Portlet and a set of portlets which can be used to annotate parts of documents with concrete media types. The media types that are currently supported are images, text and video. The Generic Annotation Portlet is the entry point in order to annotate any Information Object. If the content type of the object corresponds to one of the above, then the Generic Annotation Portlet will be able to invoke the appropriate media-specific portlet in order for the last to manage the type-specific anchored annotations of the object.

The functionality required by Annotation Front End in order to manage annotations in gCube as well as retrieve and present content (e.g. through scaling, thumbnails, type conversion etc) are offered by ASL. AFE, as every other gCube web application, stands on top of ASL by invoking its facilities in order to handle information on gCube space and access it services. AFE utilizes many of the features of ASL including session and utilities.

Among the introduced features of Annotation Front End is its support for annotating content via ontologies. The user will be given the facilities to choose between the available ontologies that are imported in gCube and then assign the annotated object to a class of the ontology.

8.4.8.1 Functions

To achieve the desired functionality Annotation Front End is formed by a set of portlets that communicate and collaborate. This set consists of the Generic Annotation Portlet and a number of portlets for every type of object (image, text, video...) that is supported for anchored annotations maintenance.

Through the generic annotation portlet the user is able to view, add, edit a non-anchored annotation. Non-anchored annotations are displayed but not the fragment of the annotated content. Delete can be performed on anchored annotations and for editing the user is redirected to the proper portlet. The annotations for an object are shown up as a list where the title of annotation, creation/modification date and author are posted up. In addition, there is the capability for the user to move to the portlet responsible for anchored annotations where he has the ability to view, add, edit and delete them.

The generic annotation portlet takes from the result set portlet the collection identifier and the object identifier. In the beginning it checks if there is an annotation collection (for non-anchored annotations) for this collection and it creates one if there isn't. Afterwards it retrieves all the non-anchored annotations connected to this object from this annotation collection and puts them to session. In addition, it takes an annotation writer/buffer for this object which is provided by the ABE Library. Every change happening in the annotations of this object is made locally in writer and session. When the user decides that he has finished his work he commits the changes and the annotations are stored through the ABE. This practice reduces the delays for managing the annotations as the ABE is invoked only two times when retrieving and storing the annotations.

Through ASL, generic annotation portlet is able to get the mimetype of the annotated object. If anchored annotations are supported for this object, generic annotation portlet takes the appropriate annotation collection for this type of object as above. Again, the annotations are loaded to the session and a writer is created for this type of anchored annotations. After that, the appropriate portlet can be invoked if the user wants to manage anchored annotations. In the end the user has to commit the changes he has made.

8.4.8.2 Graphical User Interface

The user interface provides many interesting features. Basically, it offers an editor which allows users to maintain text annotations/comments for every digital object. The portlet, apart from the text annotations, also displays their creation date and their author as well as the content of the annotated object.
In addition, it provides a set of tools which enables the creation of annotations over selections of fragments of content. The selection of a part of the Information Object is done in a user-friendly way. For example, in the case of images, the user watches the image and selects a part of it by moving and resizing a rectangle, or he may place a pin which defines a point of the image, while in text files the user selects a part of the text by dragging his mouse. Three types of multimedia content are supported for anchored annotations. These are image, text and video documents.
For some types of images that are not displayable by browsers, as tiff images, AFE utilizes the functionality of thumbnail service as exposed by ASL, in order to convert them in png format. Furthermore, AFE handles zip documents with a special manner by displaying a list with the files that they contain.
Apart from text annotations, AFE provides the capability to users to create associations between Information Objects or annotating by uploading external binary content. The first one can be performed by associating the annotating object with an object placed in basket.

Annotating via ontologies will be a new feature of Annotation Front End introduced in D4Science. The users will be able to select one of the available ontologies from a drop down list. The class hierarchy of the ontology will be displayed in a tree structure and then the users will be able to select to instantiate a class of the ontology by clicking on

the selected class and setting values on the datatype properties. In addition, object properties i.e. relationships between individuals, can be specified in order to set cross object relations.

8.4.9 Metadata Viewing Portlet

The Metadata Viewing Portlet is a graphical user interface that displays the metadata which are associated with a specific Information Object by using XSLTs defined by each user.

An Information Object may have more than one metadata objects referring to it. So, in this occasion, the portlet provides the users with the capability to choose, from a drop down list, which metadata record from each one of the available to display.

8.4.9.1 Functions

The Metadata Viewing Portlet relies to the functionality of ASL in order to retrieve the metadata of the selected object, all the available metadata schemas, the profile of the user and the presentation XSLT which the user has selected.

All the above information, which is essential for the portlet, is retrieved in parallel in order to diminish the delay in case something is not cashed. The only exception is that the presentation XSLT shall be retrieved after the user profile is read.

8.4.10 Metadata Editing Portlet

The Metadata Editing portlet is a graphical user interface that allows the user to edit the metadata associated with an Information Object. In order to handle the peculiarities of various metadata schemas and to provide highly intuitive graphical editing user interface, there have been implemented different specialized visual metadata editors for some selected popular metadata schemas, plus a generic one for generic XML schemas. The visual metadata editors enable the user to edit metadata object's elements without dealing with bare XML.

8.4.10.1 Graphical User Interface

With the visual metadata editors a user can add, edit or delete elements, when it is permitted, in a user friendly way and without demanding knowledge of XML. On the contrary, the generic editor posts the metadata in XML format and every change has to be done in the bare XML. In the generic editor, uploading of an xml file, which will replace the current metadata record, is also possible.

8.4.10.2 Functions

The Metadata Editing Portlet basically relies on the functionality of the ASL, which contacts Metadata Manager, in order to retrieve the metadata and store any changes applied to them. ASL also provides to the portlet the schema of the metadata record as well as the information if the metadata can be edited. This information exist in the profile of the metadata collection that the selected metadata object belongs to.

Having the schema of the metadata object the portlet determines which editor is the appropriate. As mentioned above there are two visual metadata editors for DC and eiDB schema and a generic one for every other schema.

8.4.11 Process Design and Monitoring Portlet

This portlet is meant for supporting the users in defining a Compound GCube Service, triggering its execution and monitoring it.

8.4.11.1 Graphical User Interface

Process design in supported by the tool in a user-friendly "graphical boxes and arrows" approach, where boxes indicate the activities of a process (for example, service invocations) and arrows symbolize the control flow. While producing process definitions is a complex activity, the user interface and the underlying process definition model

reduce the effort needed in defining a process and help to visualize it. See Appendix B.10 for a screen shot of the modelling tool.

8.4.11.2 Functions

The tool has a main area in which process can be designed by dragging and dropping boxes and arrows. The properties of the various objects are then visualized and manipulated in various areas around the main drawing canvas. The data flow of a process is designed by the concept of a process whiteboard, the in-process variable area. The whiteboard contains the global variables of a process instance. At instantiation time of a process, the whiteboard is initialized with the process input arguments. Activities consume parameters from the whiteboard and save their results back at the whiteboard. These variables can be visualized, defined and assigned in an appropriate area of the design tool. Similar areas are used to map the input and output parameters of a service invocation to the whiteboard. At the end of a process design task, the modelling tool outputs the process description in XML format and stores it in the IS. An innovative feature of the tool is the possibility to specify transactional and activation properties of the activities in a process. Each activity is described by a set of properties like execution costs, compensability, retriability, and failure probabilities. Given a transactional process description, the tool is able to check correctness of the description based on formal criteria at design time (see Section 5.6.2). For more information on the type of supportes activities refer to Section 5.6.3.

8.5 Learning Management System integration

The requirement of supporting the Learning environment usually built around a VRE led to the decision to provide Learning Management services in gCube powered infrastructures. However due to the fact that (*i*) gCube is not focusing on providing a Learning Management System (LMS) on its own, and (*ii*) powerful LMSes already exist in the open market, it has been a strategic decision to integrate gCube with an external, fully featured, open source, eLearning Management system. This integration goes up to the degree that would make the systems interoperable, without compromising the architecture or operation of any of these two systems, offering to the user a secure, comfortable to use, environment. Thus the integration is achieved at the top level of the gCube system, i.e. mostly at the presentation layer.

Moodle is one of the most popular open source LMSes. It is a fully featured, open source, e-Learning environment and a full reference to its architecture goes out of the scope of this report.

Moodle is neatly written in PHP under a GPL license and provides great customisation features while there is a plethora of extra modules, each one with its own customisation features. It is able to connect to any JDBC database and use it as a storage mechanism. Furthermore it makes use of the local file-system to store large objects. The support for SCORM packets is one of the key features of Moodle that led to its adoption.

It offers many themes that can be used in a combination with the rest of gCube platform to match users needs but most importantly it has an immense user base that makes sure that the features provided will always be up-to-date and bug free.

On the weak points of Moodle, one can count that it is not compliant with the technologies that adopted in the gCube platform. More specifically Moodle is build with PHP whereas in gCube portal the JSR168 compliant portlets are used to built the user interface inside a portal.

However a careful examination of the competition shows that there was no open-source solution compliant with these standards.

8.5.1 Moodle and gCube Interconnection

One of the most important aspects of the design is the mechanism that allows Moodle and gCube to exchange information and to interoperate in general. It is important to make clear that the Course Management System (Moodle) is a Standalone

Portal/Application that users can access directly but at the same time it can be presented through gCube Portal. However, regardless the fact that both portlets are distinct entities they must collaborate in a transparent to the user way. Needless to say that Moodle is not a gCube Portal replacement. Moodle acts as a supplement in order to satisfy the needs of e-Learning users bringing them the goods of the project.

Such a mechanism is novel in terms of technologies implemented, as the two sites (gCube and Moodle) take fundamentally different approaches in web development. gCube is comprised of JSR168 portlets that are hosted in a JSR168 enabled portal. gCube's portlets communicate with the rest of the infrastructure which is compliant with the WSRF standard. On the other hand Moodle is implemented in PHP. PHP is a scripting language with only basic support for web services. In particular Moodle makes minimal use of web services.

The WSRF standard is not implemented in PHP therefore Moodle is not able to communicate directly with the infrastructure through a PHP component. However, it is able to contact standard web services.

To combat PHP's inefficiency the following mechanism has been envisaged: Along with Moodle an extra web service implementing the interface needed to trigger any appropriate action inside Moodle scope has been implemented. This web service should be able to:

- Contact the WSRF services of the rest of D4Science infrastructure;
- Trigger appropriate actions in the Course Management System.

To manage the first point the web service is equipped with custom Java programs capable of contacting WSRF services. Towards the second point a patched source code of Moodle is used to allow the web service to directly contact Moodle's internal functions. In a typical use case scenario of the implemented mechanism:

- gCube Portal performs a standard Web service call;
- The Web service performs the action requested and returns the results, if any;
- The Web service (implemented in PHP) directly contacts Moodle or invokes a gCube Service Client;
- The Service Client (implemented in Java) is able to contact any WSRF Service of gCube;
- The results are transferred back to Moodle and the corresponding web service.

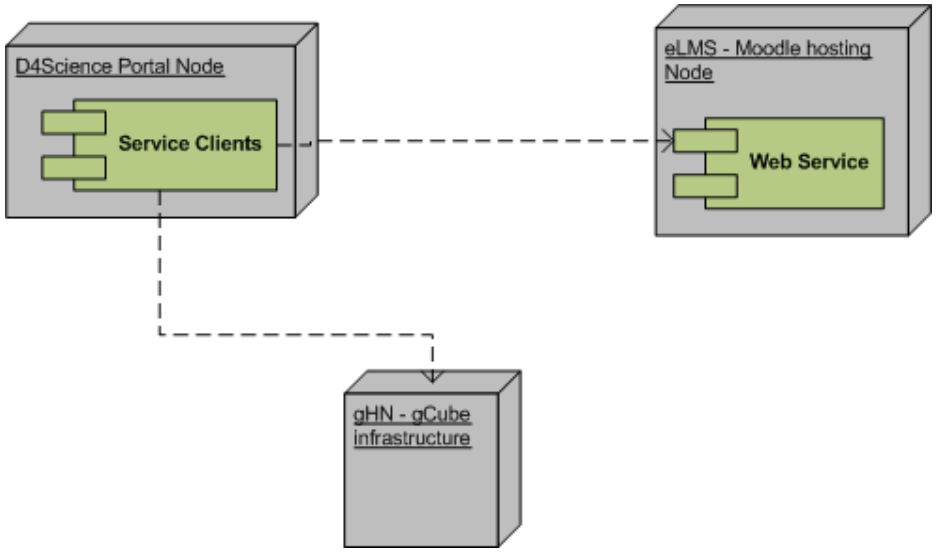

Figure 55. Moodle Integration Mechanism Architecture

The just described scheme has the following advantages:

- Easily Expandable: In order to add new functionality only a new function at the web service has to be added. If the function imposes a requirement only to Moodle then the set of service clients can stay intact. It is a very modular and easy to grasp design.
- Using Java implemented clients allows the use of the security layer of gCube.

The attempt to bridge the two portals focuses on three different fields. These fields are the bridging of the security scheme between the two portals, the content packing and content transfer between gCube - Moodle and the user interface presented to the users. These fields are described in detail in the following sections.

8.5.2 Storage

The main required functionality involves manipulating data in the context of gCube system and importing it to the Course Management System. The course of action taken to satisfy this requirement is as follows.

The concept of basket is exploited for porting content to Moodle. Each time a user conducts a search inside a gCube-powered infrastructure he/she is able to place the findings inside a basket. In this way the user is able to gather results from many search attempts. The basket can be stored and re-opened anytime the user desires to do so. In this way the user can create collections of results coming from versatile searches. However, only one basket can be active at each moment.

Using the elements stored in the active basket the user can construct packets and send them to Moodle. The implemented portlet presents the elements of the basket in a drop down menu from which the user is allowed to pick any object to place in a packet. After object selection the user chooses the lessons he wants to send the packet to and he is also required to give a name for the packet in question.

In the process of sending data to the course management system there are two completely different operations that take place:

- Gather information regarding the lessons the user is able to send data to. The user creating a packet has to be a teacher of at least one lesson in order to send the constructed packet to that specific lesson. The gCube portlet has to contact Moodle and give the identity of the user. Moodle has to respond with an array of lessons the user is able to manipulate.
- Send the data that comprise the packet. Both portals have to collaborate in order to place the packet to the correct lesson making proper authentications and authorizations.

The following figure presents the sequence of actions needed to take place.

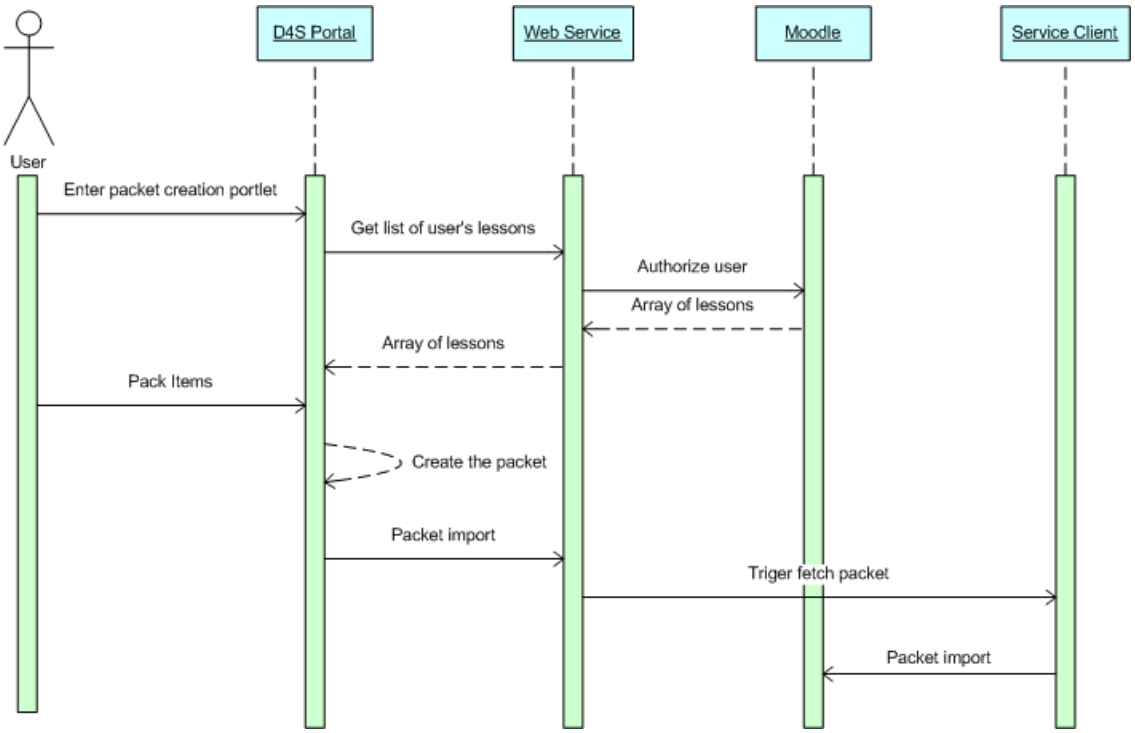

Figure 56. Packet Creation Procedure Sequence Diagram

The internal structure of the created packet is compatible with the SCORM standard. Shareable Content Object Reference Model (SCORM) is an XML-based framework used to define and access information about learning objects so they can be easily shared among different learning management systems (LMSs). SCORM was developed in response to a United States Department of Defense (DoD) initiative to promote standardization in e-learning.

The SCORM specifications, which are distributed through the Advanced Distributed Learning (ADL) Initiative Network, define an XML-based means of representing course structures, an application programming interface (API), a content-to-LMS data model, a content launch specification, and a specification for metadata records for all components of a system. The ADL specification group's next challenge is to motivate vendors to comply with SCORM specifications. Moodle directly supports this standard so the import procedure is well supported.

8.5.3 User Interface

The main goal in the intergrading the course management system with the rest of a gCube-powered infrastructure is to give the user the illusion that there is only one site.

The course management system:

- Should be accessible directly and as autonomous entity. In this way students and other interested users will be able to access their courses by passing the gCube authentication mechanism. These users are not supposed to own personal certificates and also they should not experience the delays of accessing Moodle through gCube portal.
- Should be accessible through the gCube portal in a transparent way. In this way teachers will be able to exercise the functionality provided by gCube.

In order to create the illusion that Moodle is a part of the main gCube portal a mechanism that allows for a transparent login is employed. In case a user logs in gCube portal he/she is automatically logged in Moodle as well given the fact that he owns a valid Moodle account.

The procedure of transparent login involves the following:

- the gCube portal contacts the implemented web service providing the user identity and requesting a login url;
- the web service turns to a patched version of Moodle to provide a login url;
- in case of an authorized user, the url is returned via the web service;
- the gCube portal uses the login url to present Moodle to the user and to have a cookie sent to the users browser;
- Through this cookie the user get transparently logged in to Moodle.

In Appendix B, Section B.10 you can find screenshots of the Moodle portlet.

9 CONCLUSIONS

This report focuses on documenting the overall D4Science "system" and, in particular, the gCube Framework, i.e. the software framework guaranteeing the operation of the whole system. D4Science is a complex "system" having many facets ranging from the infrastructure consisting of running software and hardware resources to the software needed to operate, or the portal through which the system is perceived by the end-users. However, all these facets result from the software system, i.e. gCube, that implements any materialisation of the D4Science "system".

The report describes the gCube software system by presenting it at three different levels of abstraction: (*i*) the *Reference Model*, by presenting an abstract framework consisting in the main concepts and their relations that characterises the system; (*ii*) the *Reference Architecture*, by presenting the architectural design pattern clarifying how the elements identified in the Reference Model have been implemented in terms of mechanisms, subsystems and relations between them; (*iii*) the *software constituents*, by presenting the design choices characterising the internals of each constituent of the Reference Architecture. The level of detail reached allows a comprehensive and clear overview of the system while hiding the implementation details that are referred to other sources of information like the deliverable DJRA4.1 Report on gCube Development or the Javadoc documentation.

This report is the first of a series of three, each expected to document the design of the "system" at different points in time during the D4Science project lifetime. The next version, i.e. DJRA1.1b, is expected at project month 12, i.e. December 2009.

APPENDIX A. GCUBE SEARCH ENGINE QUERY SYNTAX

gCube Search Engine Query Syntax

<function>	::=	<project_fun> | <sort_fun> | <filter_fun> | <merge_fun> | <join_fun> | <keeptop_fun> | <fulltexts_fun> | <fieldedsearch_fun> | <extsearch_fun> | <read_fun> | <similsearch_fun> | <spatialsearch_fun> | <retrieve_metadata_fun>
<read_fun>	::=	<read_fun_name> <epr>
<read_fun_name>	::=	'read'
<epr>	::=	string
<project_fun>	::=	<project_fun_name> <by> <project_key> <project_source>
<project_fun_name>	::=	'project'
<project_key>	::=	string
<project_source>	::=	<non_leaf_source>
<sort_fun>	::=	<sort_fun_name> <sort_order> <by> <sort_key> <sort_source>
<sort_fun_name>	::=	'sort'
<sort_key>	::=	string
<sort_order>	::=	'ASC' | 'DESC'
<sort_source>	::=	<non_leaf_source>
<filter_fun>	::=	<filter_fun_name> <filter_type> <by> <filter_statement> <filter_source>
<filter_fun_name>	::=	'filter'
<filter_type>	::=	string
<filter_statement>	::=	string
<filter_source>	::=	<non_leaf_source> | <leaf_source>

<merge_fun>	::=	<merge_fun_name> <on> <merge_sources>
<merge_fun_name>	::=	'merge'
<merge_sources>	::=	<merge_source> <and> <merge_source> <merge_sources2>
<merge_sources2>	::=	<and> <merge_source> <merge_sources2>
<merge_source>	::=	<left_parenthesis> <function> <right_parenthesis>
<join_fun>	::=	<join_fun_name> <join_type> <by> <join_key> <on> <join_source> <and> <join_source>
<join_fun_name>	::=	'join'
<join_key>	::=	string
<join_type>	::=	'inner' | 'fullOuter' | 'leftOuter' | 'rightOuter'
<join_source>	::=	<left_parenthesis> <function> <right_parenthesis>
<keeptop_fun>	::=	<keeptop_fun_name> <keeptop_number> <keeptop_source>
<keeptop_fun_name>	::=	'keeptop'
<keeptop_number>	::=	integer
<keeptop_source>	::=	<non_leaf_source>
<fulltexts_fun>	::=	<fulltexts_fun_name> <by> <fulltexts_term> <fulltexts_terms> <in> <language> <on> <fulltexts_sources>
<fulltexts_fun_name>	::=	'fulltextsearch'
<fulltexts_terms>	::=	<comma> <fulltexts_term> <fulltexts_terms>
<fulltexts_sources>	::=	<fulltexts_source> <fulltexts_sources_2>
<fulltexts_sources_2>	::=	<comma> <fulltexts_source> <fulltexts_source>
<fulltexts_source>	::=	string
<fieldedsearch_fun>	::=	<fieldedsearch_fun_name> <by> <query> <fieldedsearch_source>

<fieldedsearch_fun_name>	::=	'fieldedsearch'
<query>	::=	string
<fieldedsearch_source>	::=	<non_leaf_source> | <leaf_source>
<extsearch_fun>	::=	<extsearch_fun_name> <by> <extsearch_query> <on> <extsearch_source>
<extsearch_fun_name>	::=	'externalsearch'
<extsearch_query>	::=	string
<extsearch_source>	::=	string
<similsearch_fun>	::=	<similaritysearch_fun_name> <as> <URL> <by> <pair> <pairs> <similarity_source>
<similsearch_fun_name>	::=	'similaritysearch'
<URL>	::=	string
<pair>	::=	<feature> <equal> <weight>
<pairs>	::=	<and> <pair> <pairs> | Ï†
<similarity_source>	::=	<leaf_source>
<if-syntax>	::=	<if> <left_parenthesis> <function-st> <compare-sign> <function-st> <right_parenthesis> <then> <search-op> <else> <search-op>
<compare-sign>	::=	'==' | '>' | '<' | '>=' | '<='
<function-st>	::=	<left-op> <math-op> <right-op>
<math-op>	::=	'+' | '-' | '*' | '/'
<left-op>	::=	<function> <left_parenthesis> <left-op> <right_parenthesis> | <literal>
<function>	::=	<max-fun> | <min-fun> | <sum-fun> | <av-fun> | <var-fun> | <size-fun>
<max-fun>	::=	'max' <left_parenthesis> <xpath> <comma> <search-op> <right_parenthesis>
<min-fun>	::=	'min' <left_parenthesis> <xpath> <comma> <search-op> <right_parenthesis>
<sun-fun>	::=	'sum' <left_parenthesis> <xpath> <comma> <search-op> <right_parenthesis>

<av-fun>	::=	'av' <left_parenthesis> <xpath> <comma> <search-op> <right_parenthesis>
<var-fun>	::=	'var' <left_parenthesis> <xpath> <comma> <search-op> <right_parenthesis>
<size-fun>	::=	'size' <left_parenthesis> <search-op> <right_parenthesis>
<right-op>	::=	<function-st> | <left-op>
<xpath>	::=	'<field selection through xpath>'
<retrieve_metadata_fun>	::=	<rm_fun_name> <in> <language> <on> <rm_source> <as> <schema>
<rm_fun_name>	::=	'retrievemetadata'
<schema>	::=	string
<rm_source>	::=	<left_parenthesis> <function> <right_parenthesis>
<spatialsearch_fun>	::=	<spatialsearch_fun_name> <relation> <geometry> [<timeBoundary>] <spatial_source>
<spatialsearch_fun_name>	::=	'spatialsearch'
<relation>	::=	{'intersects', 'contains', 'isContained'}
<geometry>	::=	<polygon_name> <left_parenthesis> <points> <right_parenthesis>
<timeBoundary>	::=	'within' <startTime> <stopTime>
<startTime>	::=	double
<stopTime>	::=	double
<spatial_source>	::=	<leaf_source>
<points>	::=	<point> {<comma> <point>}+
<x>	::=	integer
<y>	::=	integer
<leaf_source>	::=	[<in> <language>] <on> <source> [<as> <schema>]
<non_leaf_source>	::=	<left_parenthesis> <function> <right_parenthesis>
<language>	::=	'AFRIKAANS' | 'ARABIC' | 'AZERI' | 'BYELORUSSIAN' | 'BULGARIAN' | 'BANGLA' | 'BRETON' | 'BOSNIAN' |

		'CATALAN' | 'CZECH' | 'WELSH' | 'DANISH' | 'GERMAN' | 'GREEK' | 'ENGLISH' | 'ESPERANTO' | 'SPANISH' | 'ESTONIAN' | 'BASQUE' | 'FARSI' | 'FINNISH' | 'FAEROESE' | 'FRENCH' | 'FRISIAN' | 'IRISH_GAELIC' | 'GALICIAN' | 'HAUSA' | 'HEBREW' | 'HINDI' | 'CROATIAN' | 'HUNGARIAN' | 'ARMENIAN' | 'INDONESIAN' | 'ICELANDIC' | 'ITALIAN' | 'JAPANESE' | 'GEORGIAN' | 'KAZAKH' | 'GREENLANDIC' | 'KOREAN' | 'KURDISH' | 'KIRGHIZ' | 'LATIN' | 'LETZEBURGESCH' | 'LITHUANIAN' | 'LATVIAN' | 'MAORI' | 'MONGOLIAN' | 'MALAY' | 'MALTESE' | 'NORWEGIAN_BOKMAAL' | 'DUTCH' | 'NORWEGIAN_NYNORSK' | 'POLISH' | 'PASHTO' | 'PORTUGUESE' | 'RHAETO_ROMANCE' | 'ROMANIAN' | 'RUSSIAN' | 'SAMI_NORTHERN' | 'SLOVAK' | 'SLOVENIAN' | 'ALBANIAN' | 'SERBIAN' | 'SWEDISH' | 'SWAHILI' | 'TAMIL' | 'THAI' | 'FILIPINO' | 'TURKISH' | 'UKRAINIAN' | 'URDU' | 'UZBEK' | 'VIETNAMESE' | 'SORBIAN' | 'YIDDISH' | 'CHINESE_SIMPLIFIED' | 'CHINESE_TRADITIONAL' | 'ZULU'
<source>	::=	string
<schema>	::=	string
<left_parenthesis>	::=	'('
<right_parenthesis>	::=	')'
<comma>	::=	','
<and>	::=	'and'
<on>	::=	'on'
<as>	::=	'as'
<by>	::=	'by'
<sort_by>	::=	'sort'
<from>	::=	'from'
<if>	::=	'if'
<then>	::=	'then'
<else>	::=	'else'

APPENDIX B. USER INTERFACE

B.1. Search Portlets

Figure 57 shows the User Interface of the collections navigator portlet.

Figure 57. Collections Navigator Portlet

A full description for each collection is available by pressing the "more" button. Figure 58 shows this functionality.

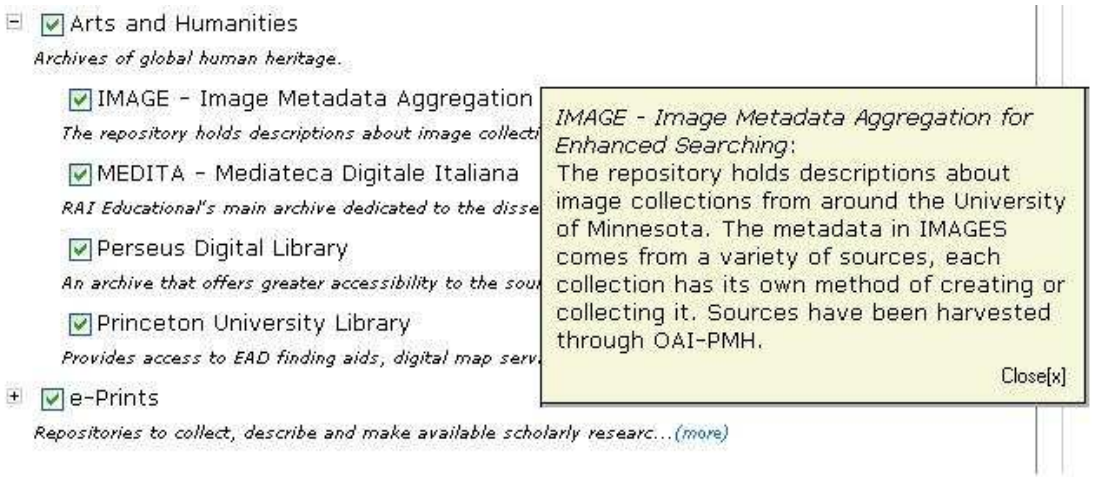

Figure 58. Collection's Description

Figure 59 shows the simple search portlet. A search for the given keywords is performed on the selected collections.

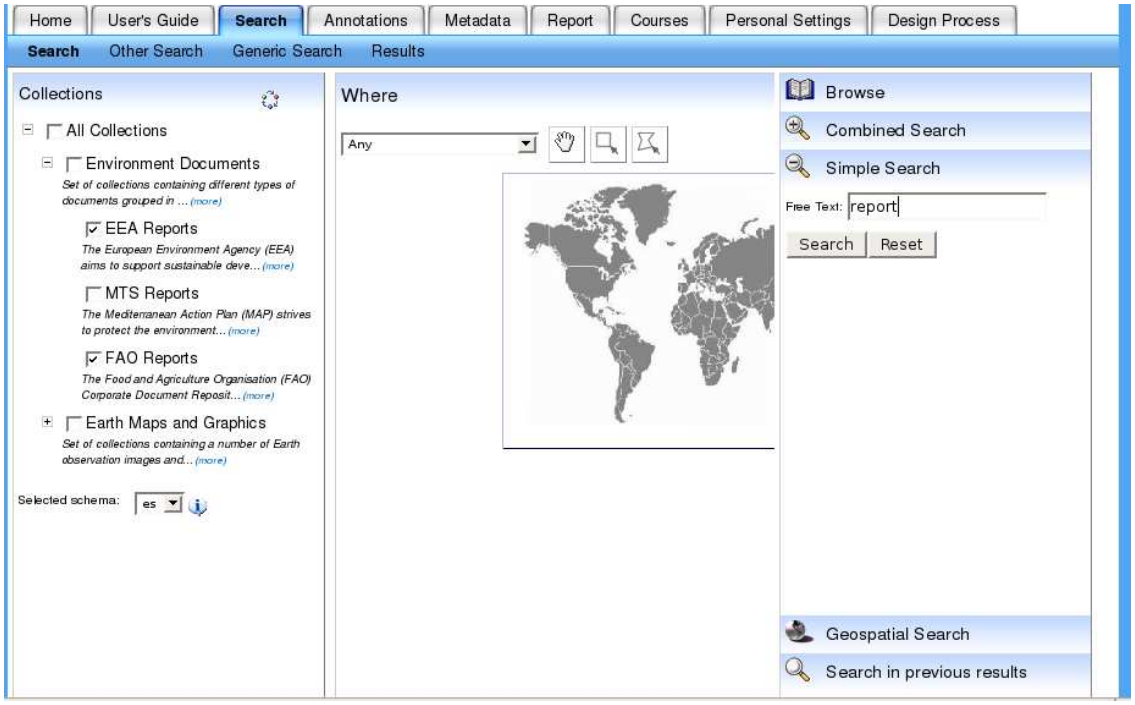

Figure 59. Simple Search Portlet

Figure 60 shows the browse portlet. Browse by different fields can be performed on the selected collections. The results can be displayed by aascending or descending order, depending on user's selection.

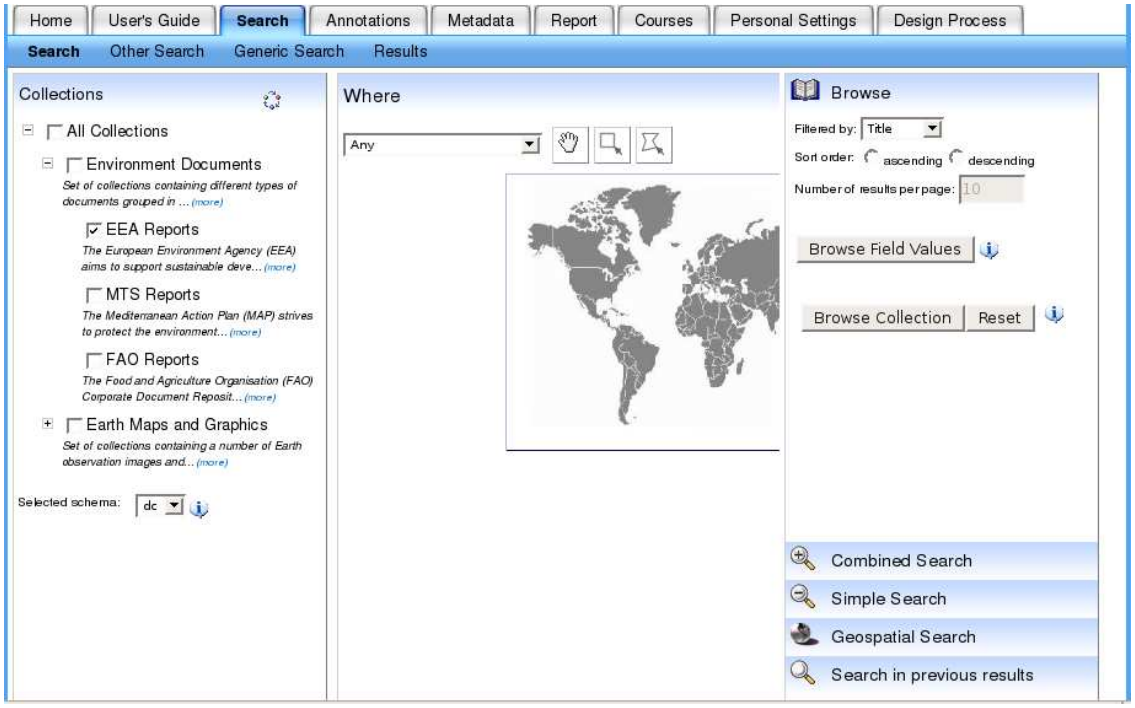

Figure 60. Browse Portlet

Figure 61 shows the combined search portlet. As Figure 62 shows, the query of the combined search can contain more than one condition depending on what the user wants to search for.

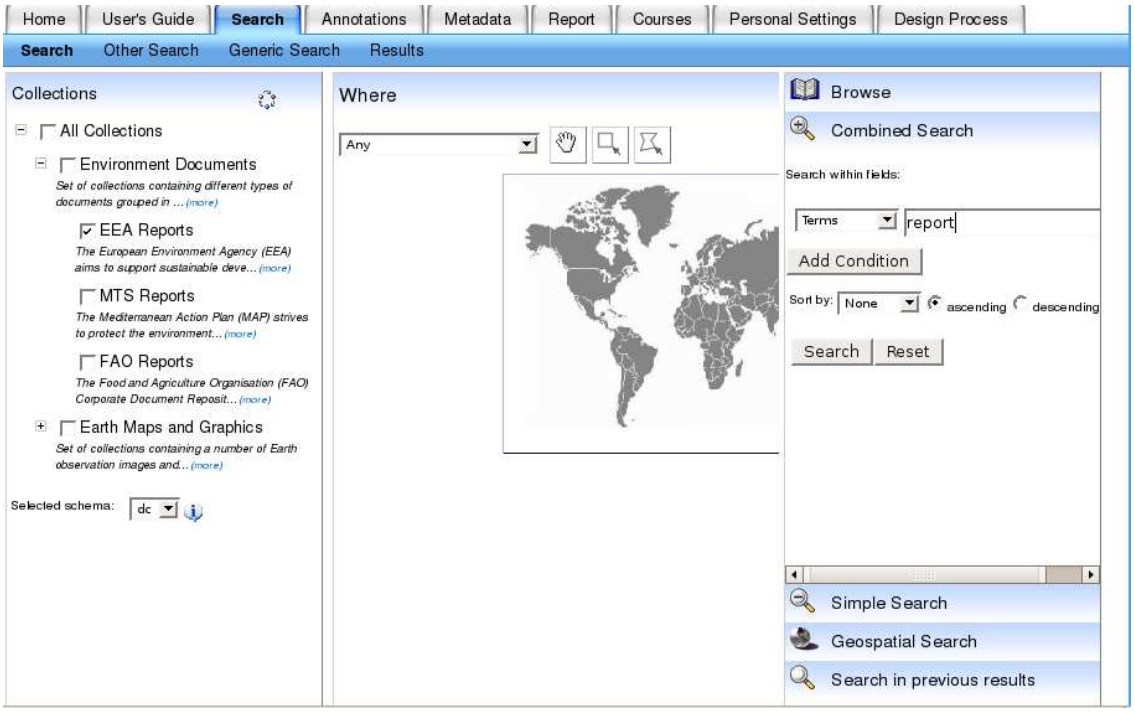

Figure 61. Combined Search Portlet

Figure 62. Combined Search Portlet - Search Conditions

Figure 63 shows the previous results portlet. After selecting which previous query you want to refine, then you can add the keywords to search for.

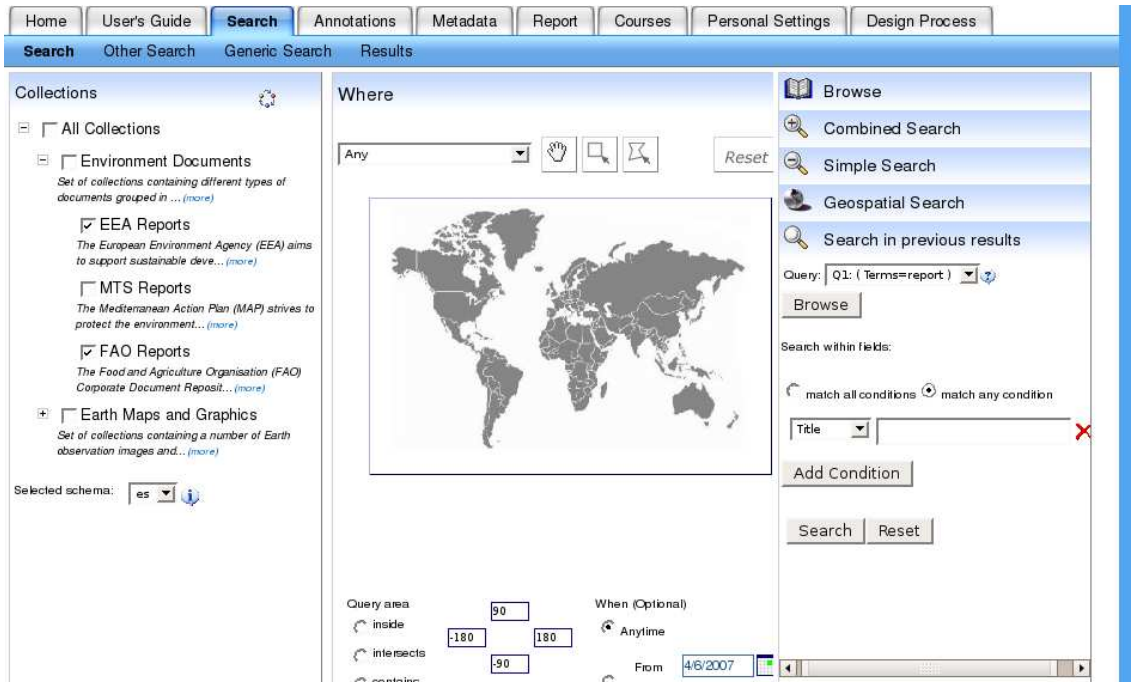

Figure 63. Search Previous Results Portlet

B.2. Quick and Google Search Portlet

Figure 64 shows the quick and google search portlet. In the text area users write the keywords they want to search for.

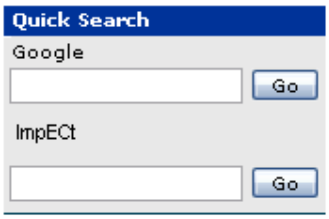

Figure 64. Quick and Google Search

B.3. Browse Results Portlet

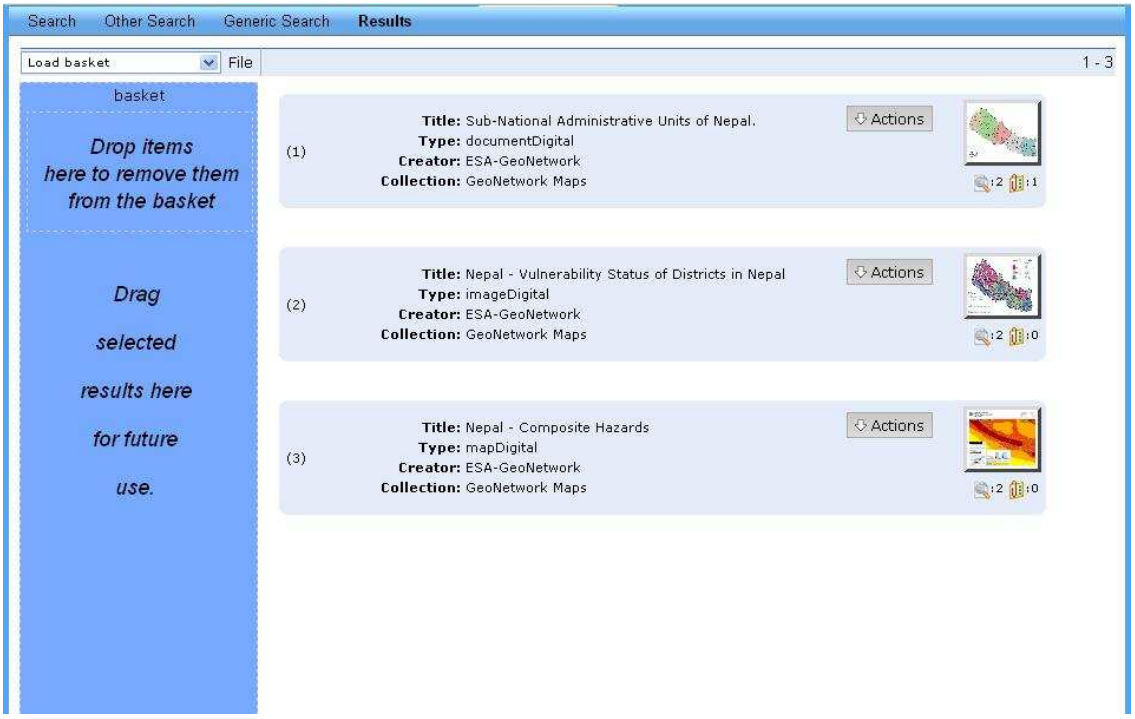

Figure 65. Browse Results - Layout

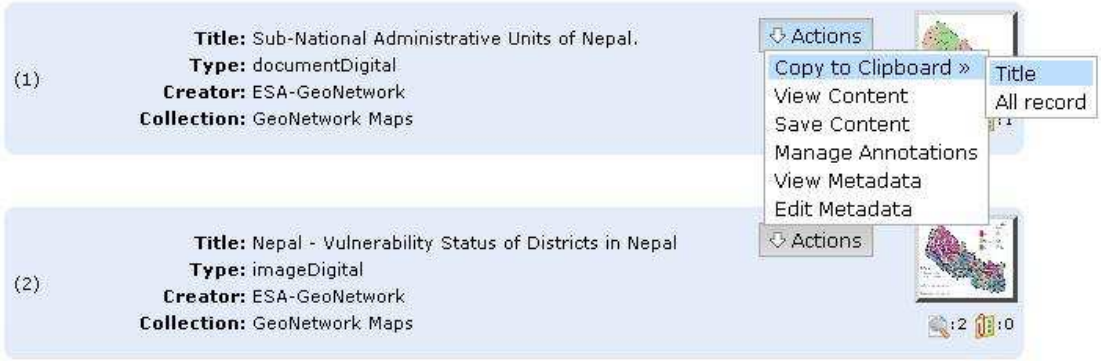

Figure 66. Browse Results - Available Actions Menu

B.4. User Profile Editing Portlet

Figure 67 shows the user profile editing portlet. Some information (e.g the user's full name) cannot change but other information can be changed by the user.

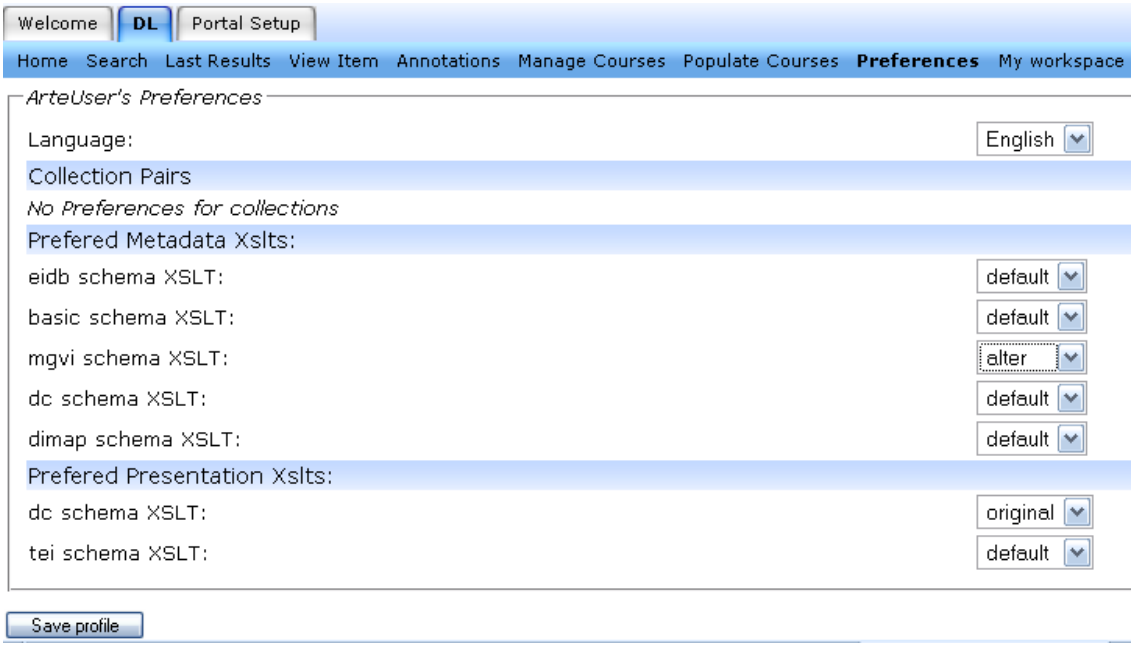

Figure 67. User Profile Editing Portlet

B.5. Profile Administration Portlet

Figure 68 shows the profile administration portlet.

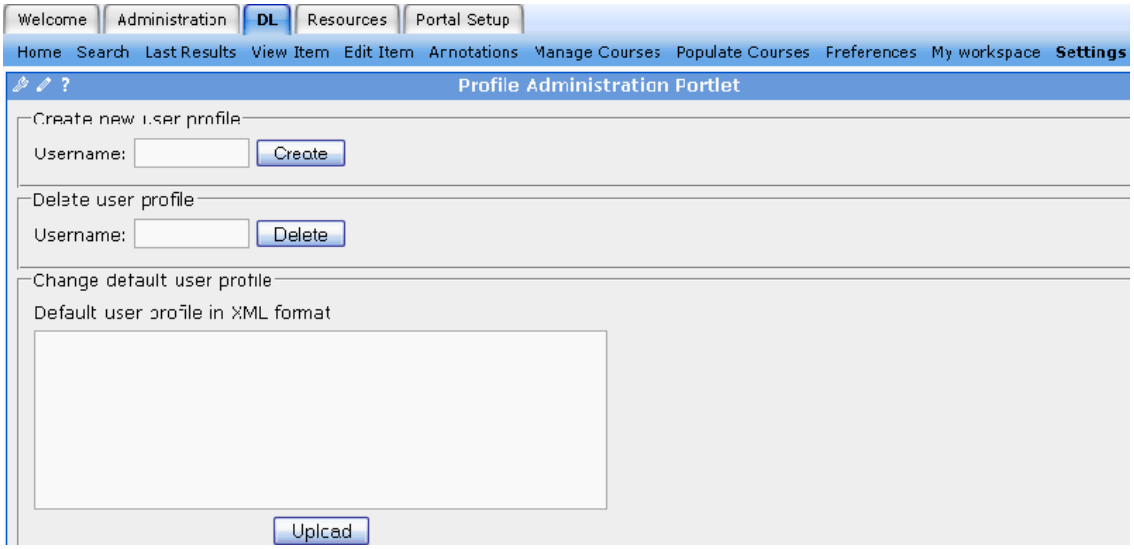

Figure 68. Profile Administration Portlet

B.6. Metadata Broker Administration Portlet

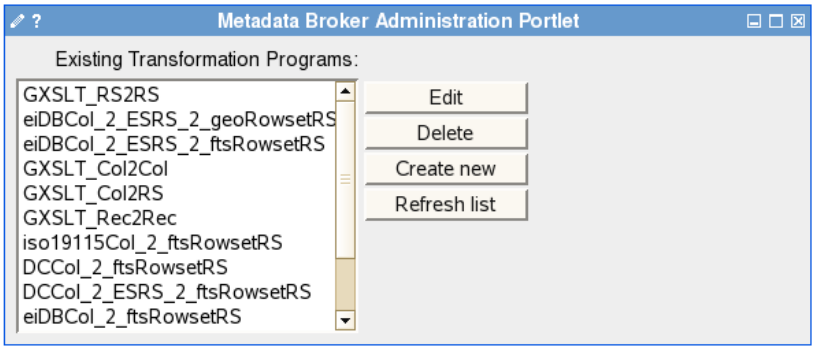

Figure 69. Metadata Broker Administration Portlet

A screenshot of the Metadata Broker Administration portlet in edit mode is shown in Figure 70.

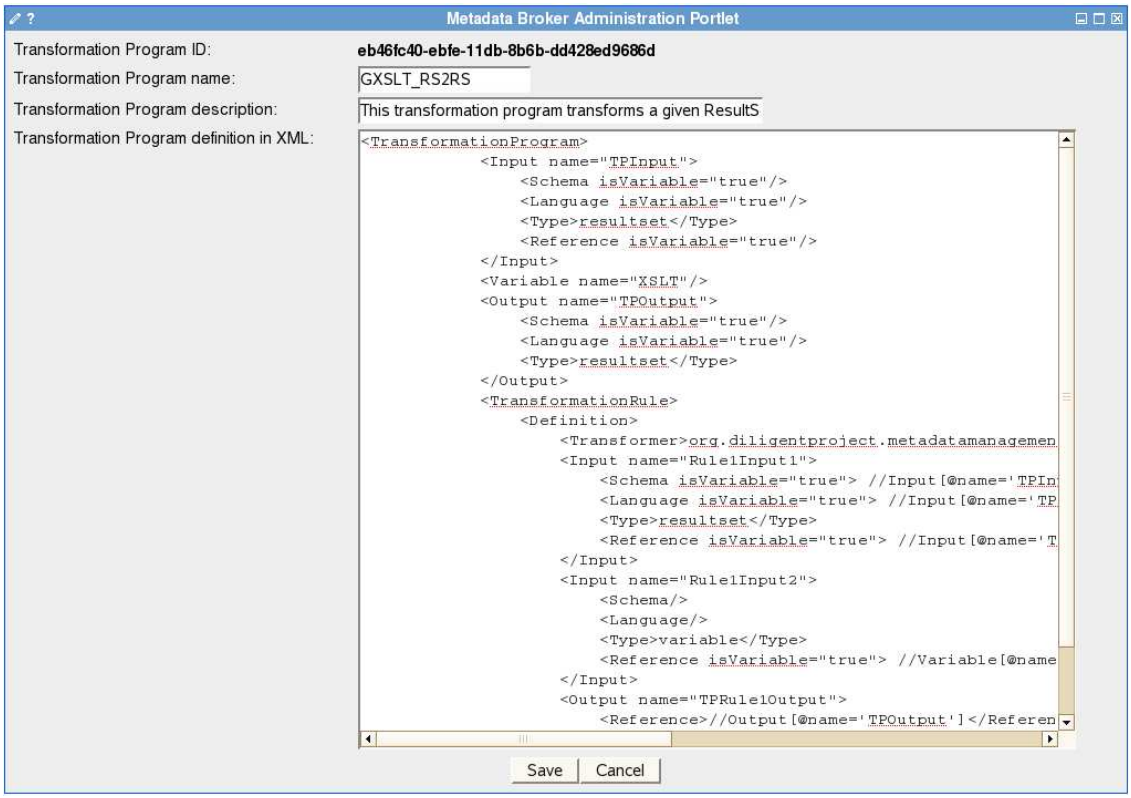

Figure 70. Metadata Broker Transformation Program Editor

B.7. Annotation Front End Portlets

The next Figure depicts a generic Annotation Porlet

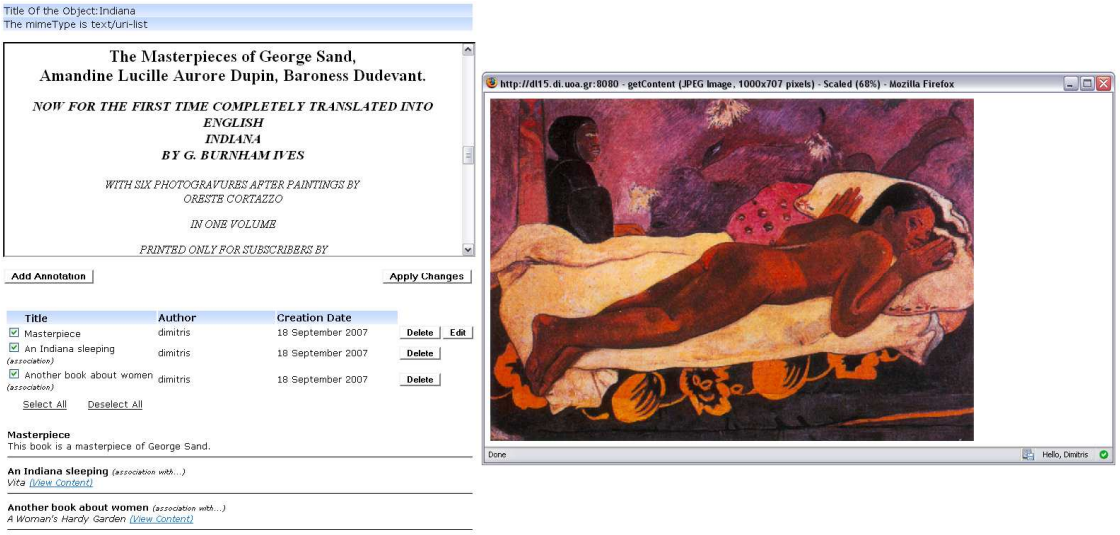

Figure 71: Generic Annotation Portlet

The next Figure depicts an Image Annotation Porlet, where anchored annotations are placed as boxes around areas of the image.

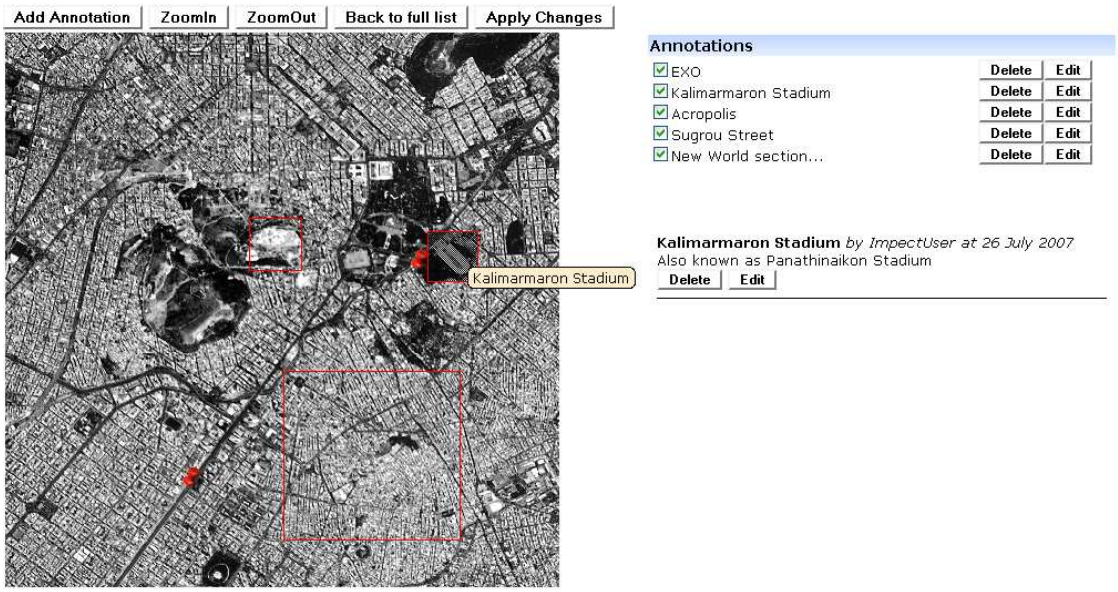

Figure 72. Image Annotation Portlet

The next Figure depicts an Text Annotation Porlet, where anchored annotations are placed in selections of text.

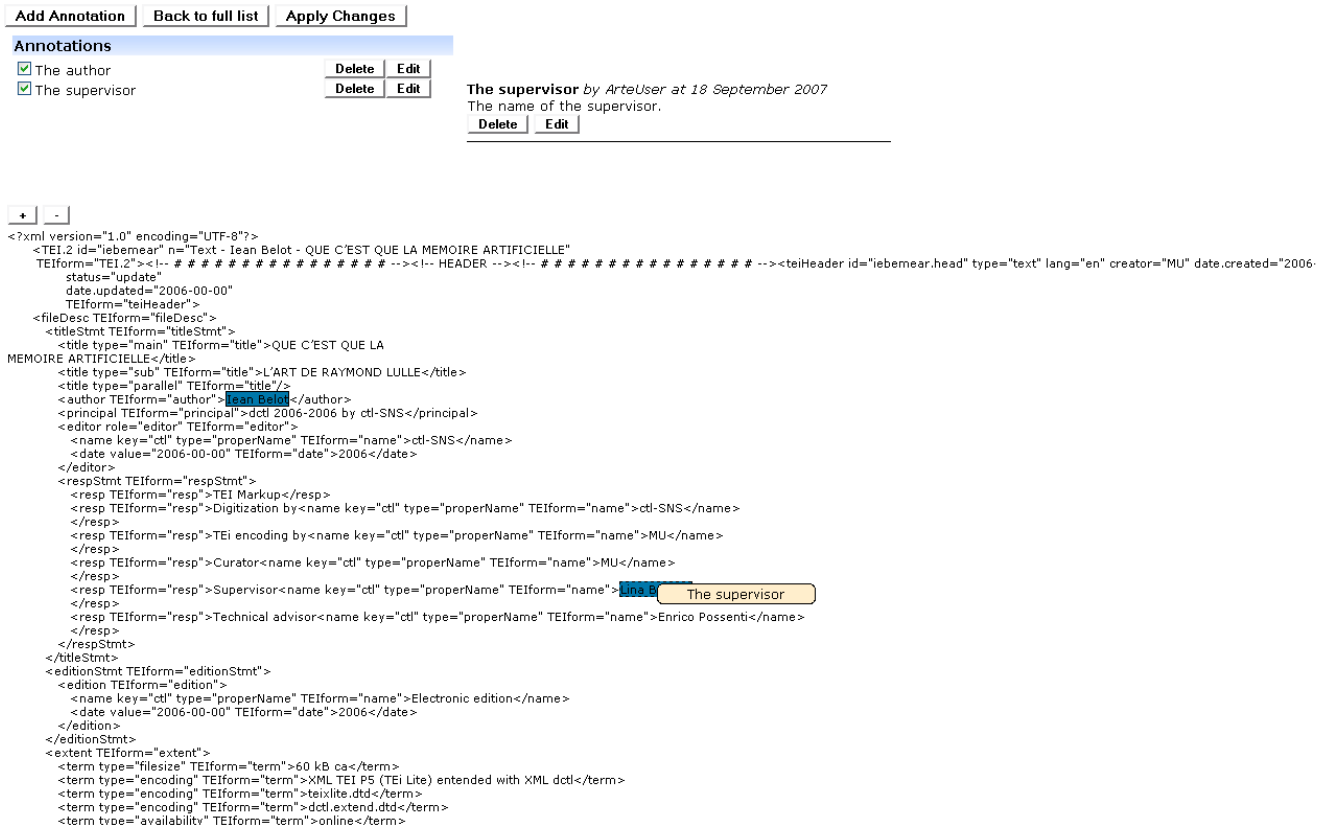

Figure 73. Text Annotation Portlet

B.8. Metadata Viewing Portlet

The next picture depicts the MetadataViewing portlet which projects XML metadata into browser friendly HTML.

Image ID
657
Title
Faliro Olympic Arena, Athens, Greece
Caption
The different stages of construction of the Faliro Olympic complex can be seen in the reddish pixels of this detail of a ERS-2 Synthetic Aperture Radar (SAR) multitemporal colour composite image. The complex is located on the bay of Faliro, 8 km south of Athens, to the southeast of Pireas. Construction of the complex started in June 2002 and was completed in March 2004. The three Precision Radar Image Images that make up this colour composite were acquired over 2003-2004 and thus show in reddish colour the changes in ground profile sensed by the SAR instrument, among which are the appearance of the 8,536-seat Faliro arena which will host the handball and tae-kwon-do competitions at the 2004 Olympics.
Width
Center Latitude
37.80
Center Longitude
24.15
File Reference
Athens_Greece_Faliro_SAR_IM_Orbit_45910_20040131.tif
Credits
European Space Agency. All rights reserved.
Satellite
ERS-2
Instrument
SAR
Resolution
Time period
Sensed 2003-03-31
Extras
Group Europe Category Countries Keyword Greece
Group Europe Category Cities Keyword Athens - Greece

Figure 74. Metadata Viewing Portlet

B.9. Metadata Editing Portlet

A schema aware metadata editing portlet is depicted below (DC schema)

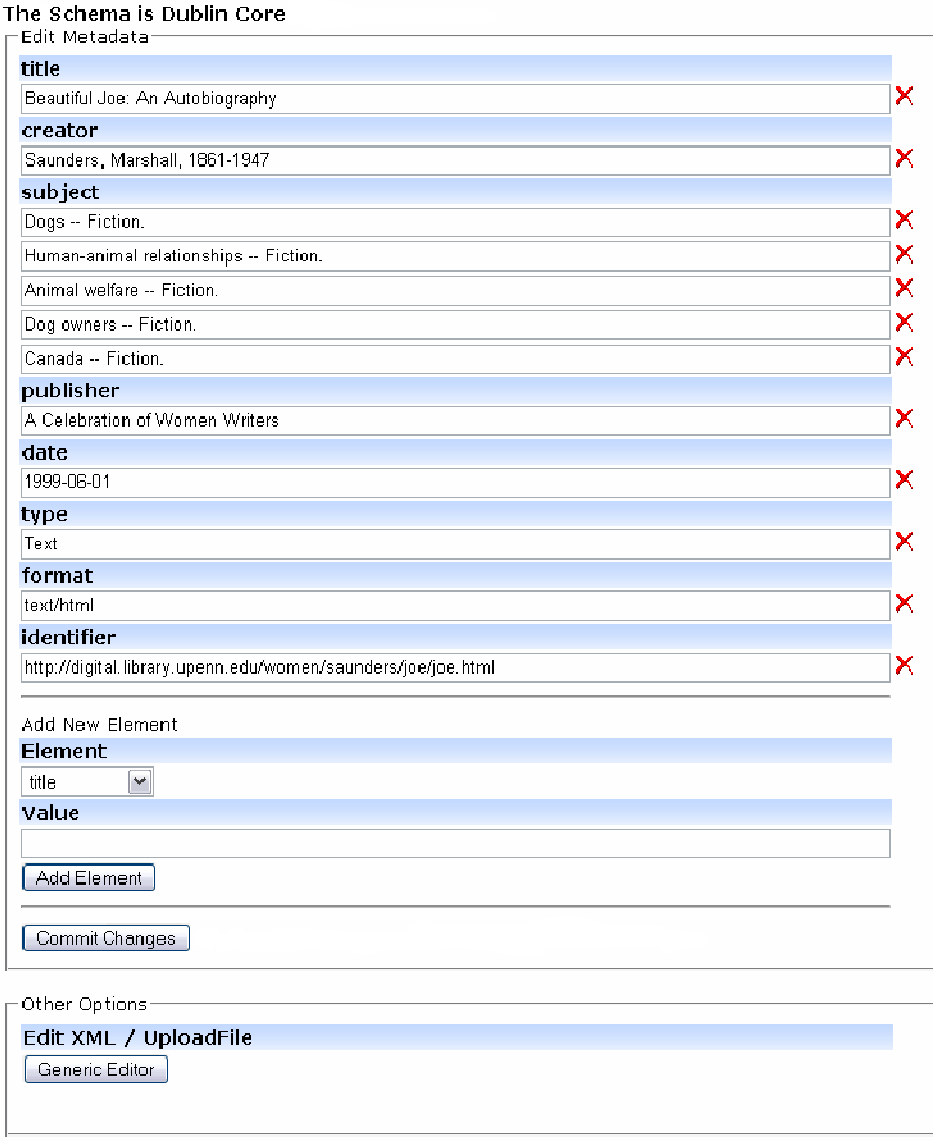

Figure 75. Visual Editor for Dublin Core Schema

The following schema depicts a generic Metadata Editor which handles metadata as text (XML).

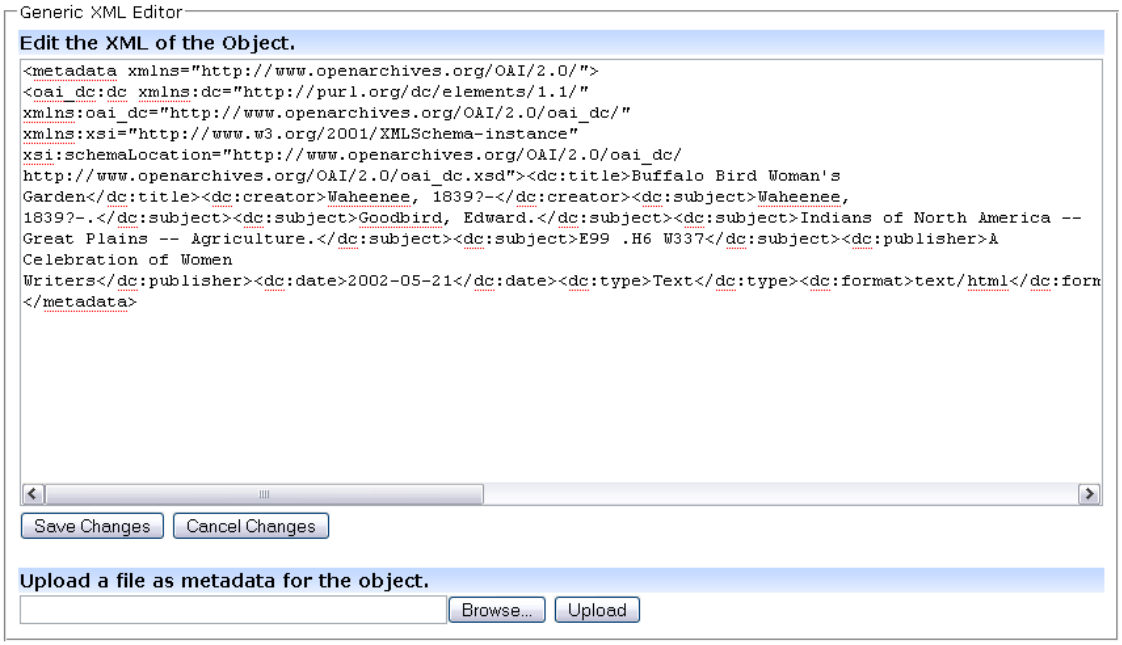

Figure 76. Generic Metadata Editor

B.10. Process Design and Monitoring Portlet

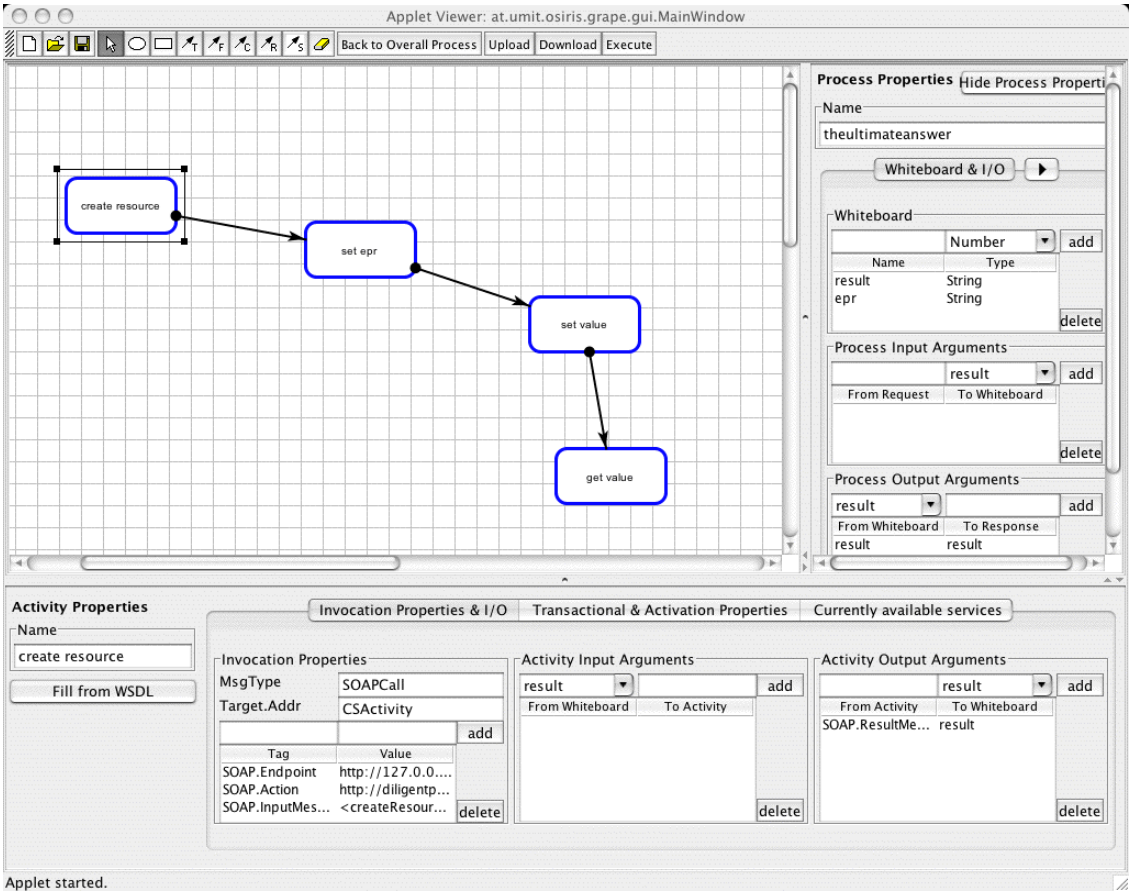

Figure 77. Process Design Modelling Tool

B.11. Moodle Portlet

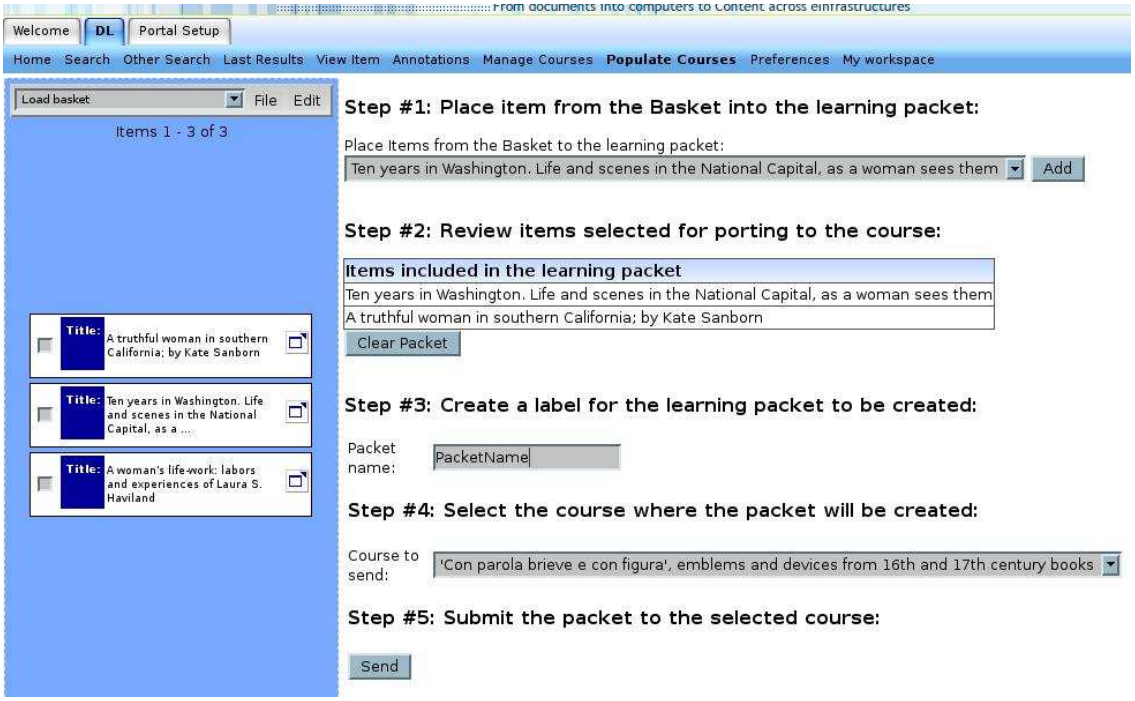

Figure 78. SCORM Packet Creation Portlet

The following figures present different views of the same Moodle site. The course management site is presented inside a frame in the gCube portal and, additionally, the same site is a stand alone portal.

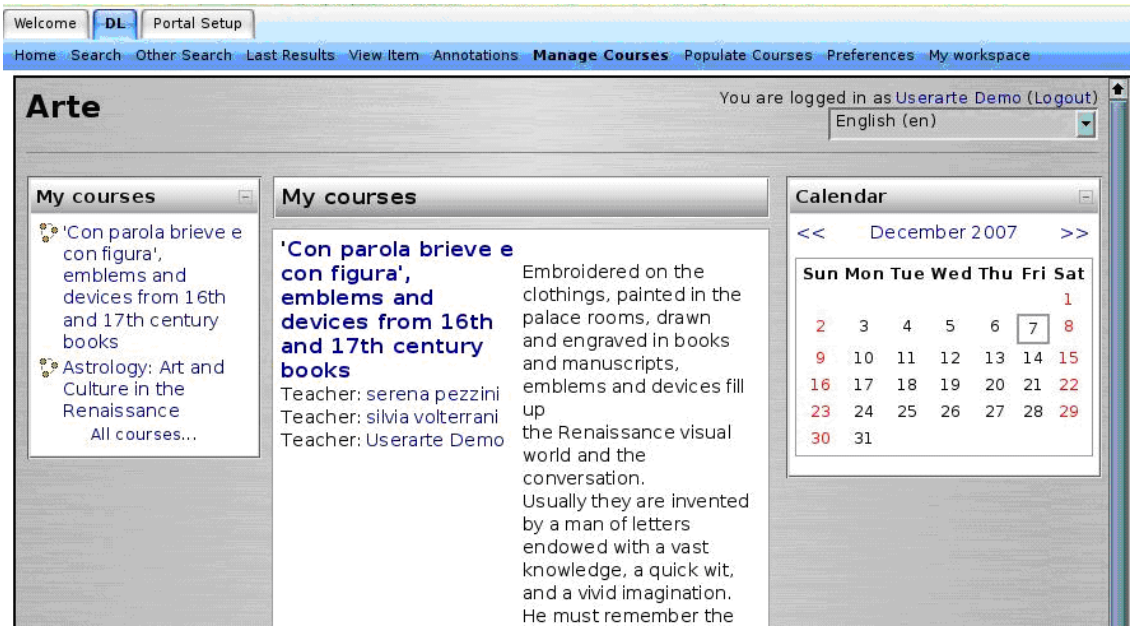

Figure 79. Moodle Portal - Standalone View

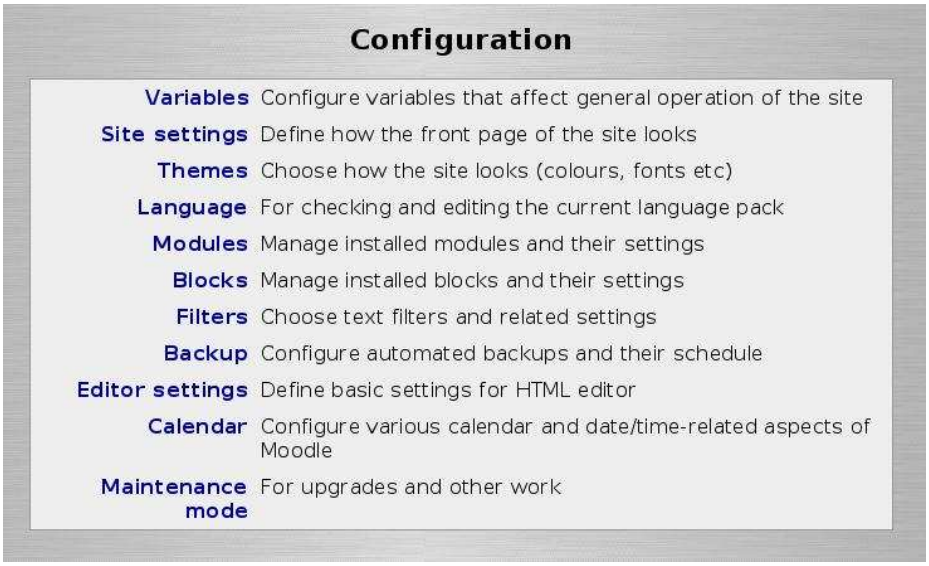

Figure 80. gCube Portal - Moodle's Configuration

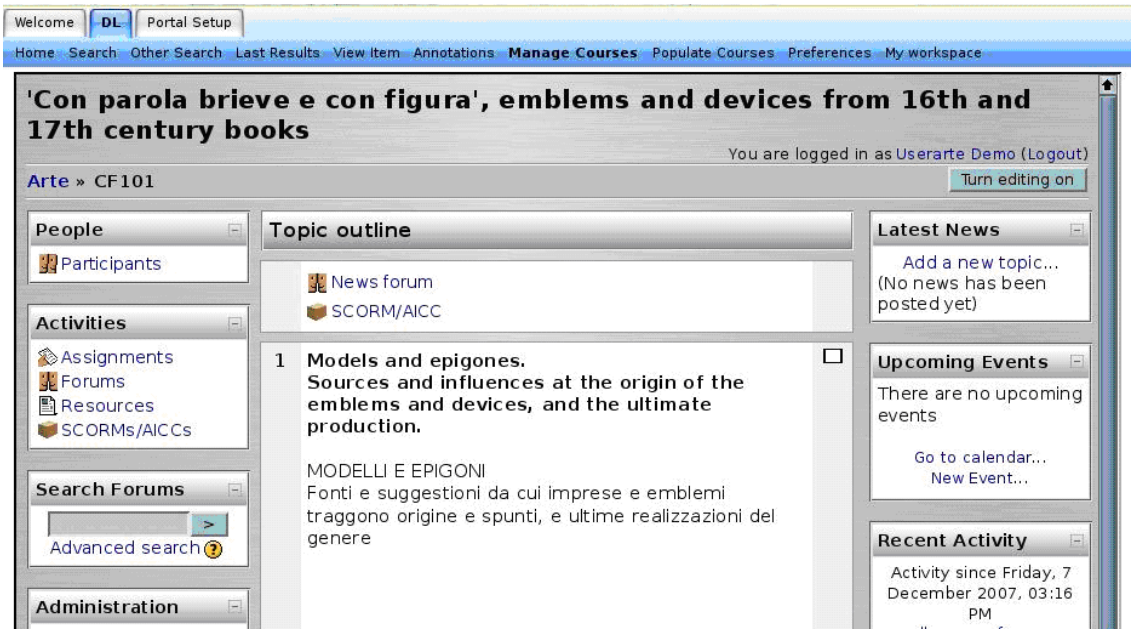

Figure 81. Moodle's Course View

GLOSSARY

AAA

Short for *Authentication, Authorization* and *Accounting*, a system to control what resources users have access to and to keep track of what activity they perform on them.

Accounting

The process of keeping track of a user's activity while accessing the resources, including the amount of time spent in their usage, the services accessed while there and the amount of data transferred during the session. Accounting data are used for trend analysis, capacity planning, billing, auditing, and cost allocation.

Authentication

The process of identifying a user usually based on username and password. Authentication is based on the idea that each individual user has a unique piece of information that sets him or her apart from other users.

Authorization

The process of granting or denying a user access to resources once the user has been authenticated through the appropriate authentication mechanism, e.g. username and password. The amount of information and the amount of services the user has access to depend on the user's authorization level.

Certification Authority

A trusted third-party organization or company that issues digital certificates used to create digital signatures and public-private key pairs. The role of the CA in this process is to guarantee that the individual granted the unique certificate is, in fact, who he or she claims to be.

Collection

A set of *Information Object*, which is in turn an *Information Object*, The extension of a collection consists of the *Information Objects* it contains. A collection may be defined by a membership criterion, which is the intension of the collection.

Component

A constituent of a larger whole. In the gCube context is a reusable block of software (code, metadata and other resources, depending on the technology) consisting of one or more sub-elements, often strongly interrelated, that encloses a specific (usually small) domain of functionality. Components can be consumed / distributed in libraries or executables (standalone or services) and offer technology dependent interfaces for their consumption.

eInfrastructure

The term e-Infrastructure refers to the new research environment in which all researchers - whether working in the context of their home institutions or in national or multinational scientific initiatives - have shared access to unique or distributed scientific facilities (including data, instruments, computing and communications), regardless of their type and location in the world. (CORDIS, http://cordis.europa.eu/fp7/ict/e-infrastructure/)

Framework

A set of software (compile/run-time elements, libraries, tools, services), documentation, policies, procedures, that supports the implementation of technology specific higher-level software elements.

gCore Framework (gCF)

A java-based application framework for (*i*) building high-quality gCube services that are easily deployable both via the infrastructure facilities and to a human and (*ii*) providing a runtime environment for such services. The framework is distributed with gCore.

gCube

gCube stands for Grid for Hardware, Data and Software. It is a multifaceted system as reflected by the definition below.

A distributed service-based system for the autonomic operation of an e-Infrastructure for Virtual Research Environments.

A distributed runtime platform for the development of interactive, service based e-Science applications.

A full-featured, expandible, professional-grade and service-based platform that is sufficiently functional to create a broad range of dynamic production-ready applications (VREs).

gCube Core Distribution (gCore)

A packaging of the components of the gCube system which are ubiquitously deployed on each gHN of a gCube infrastructure.

gCube Hosting Node (gHN)

The topological unit of a gCube infrastructure. An abstraction over a container running on a given port and hosting at least a minimal set of basic gCube services (the local services) dedicated to the host management.

gLite

gLite (pronounced "gee-lite") is a middleware for Grid computing. Born as a part of the EGEE Project, gLite provides a bleeding-edge, best-of-breed framework for building Grid applications tapping into the power of distributed computing and storage resources across the Internet. [22]

Information Object

A logical unit of information potentially consisting of and linked to other Information Objects as to form *compound objects.*

Information Retrieval

Information retrieval (IR) is the act/science of searching for documents, for information within documents and for metadata about documents, as well as that of searching relational databases and the World Wide Web. (Wikipedia, August 2008, http://en.wikipedia.org/wiki/Information_retrieval)

Library (Software Library)

A reusable set of (locally) executable/embedable software (code, metadata and other resources, depending on the technology) that encloses one or more components and (usually) is not standalone usable. Libraries are packaged as indentifiable "file system" elements, offer technology dependent interfaces for their consumption and are consumed locally.

Resource Provider (Resource Owner)

The entity who owns a resource and thus has full privileges in using it.

Search Engine

A search engine encompasses all the tools and functionalities for enabling and perfoming lookups for information in pools of data / information. As such it might contain functional (for supporting various types of searches, processing/transforming the data, performing natural language processing etc) and non-functional (for enhancing performance, security, availability and reliability) elements of a large range of funtionalities: indexing, feature extraction, optimizer, feature-distance calculator, XML processing/filtering, visualisation etc. Harvesting / updating and storage handling is considered a boundary operation for the Search Engine in D4Science. A search engine is not mandatorily bound to web searching, and can often, in simple scenarios, be implemented over a single index (such as a FullText one).

Service

A *standalone executable* piece of software that offer its functionality to other executable software. Services do not involve human intervention for their execution and often adopt a technology independent communication protocol. Services can be locally ore remotelly consumed and usually support network discovery and consumption.

Service Oriented Architecture (SOA)

The approach of composing large systems out of looselly coupled services, which expose well defined interfaces for their invocation. Service Oriented Architecture is orthogonal to Component Oriented development and Object Oriented Programming.

Standalone Executables

Software, build on a technology-specific framework, that can consumes libraries, services and other resources that can ultimatelly execute and offer functionality on its own. Standalone executables can act as desktop/user applications or services themselves

Subsystem

A part of a system that can be considered a system by its own.

System

A set of connected things or parts forming a complex whole.

Virtual Organisation

A virtual organisation (VO) is a dynamic pool of distributed resources shared by a dynamic set of users belonging to one or more organizations in a trusted way.

Virtual Research Environment (VRE)

An integrated and coordinated working environment providing participants with the resources (data, instruments, processing power, communication tools, etc.) they need to accomplish the envisaged task. The resources shared can be of very different nature and vary across application and institutional domains. Usually they include *content resources*, *application services* that manipulate these content resources to produce new knowledge, and *computational resources,* which physically store the content and support the processing of the services.

From a system point of view, a VRE is a pool of gCubeResources dynamically aggregated to behave as a unit w.r.t. the application context the VRE is expected to serve. Each VRE is a view over the potentially unlimited pool of resources made available through the Infrastructure that (*i*) is regulated by the user community needs and the resources

sharing policies and (*ii*) produces a new VO constraining the scope and usage of resources actors playing in the VRE are subject to.

Web Service

A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

REFERENCES

[1] Alfieri, R., Cecchini, R., Ciaschini, V., dell'Agnello, L., Frohner, Á., Lőrentey, K., and Spataro, F. *From gridmap-file to VOMS: managing authorization in a Grid environment*. Future Gener. Comput. Syst. 21, 4 (Apr. 2005), 549-558.

[2] Andrade, P.; Candela, L.; Pagano, P.; Simi, M. Architectural Specification. DILIGENT Deliverable N. D1.1.2

[3] Antonioletti, M.; Hastings, S.; Krause, A.; Langella, S.; Lynden, S.; Laws, S.; Malaika, S.; Paton, N. Web Services Data Access and Integration – The XML Realization (WS-DAIX) Specification, Version 1.0. Open Grid Forum DocumentGFD.75. August 2006

[4] Apache Maven Project. [Online]. http://maven.apache.org

[5] Avancini, H.; Borriello, M.; Candela, L.; Fabriani, P.; Manzi, A.; Pagano, P.; Roccetti, P.; Simi, M.; Turli, A. *DL Creation and Management Services Detailed Design Report*. DILIGENT Deliverable N. D1.2.3

[6] Banks T. Web Services Resource Framework (WSRF) – Primer v1.2. OASIS Committee Draft, 2006 http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf

[7] Bennett, K.; Layzell, P.; Budgen, D.; Brereton, P.; Macaulay, L.; Munro, M. *Service-based software: the future for flexible software*. In Seventh Asia-Pacific Software Engineering Conference (APSEC'00), 2000

[8] Boag, S.; Chamberlin, D.; Fernández, M.F.; Florescu, D.; Robie, J.; Siméon, J. *XQuery 1.0: An XML Query Language*. W3C Recommendation. http://www.w3.org/TR/xquery/

[9] Box, D.; Christensen, E.; Curbera, F.; Ferguson, D.; Frey, J.; Hadley, M.; Kaler, C.; Langworthy, D.; Leymann, F.; Lovering, B.; Lucco, S.; Millet, S.; Mukhi, N.; Nottingham, M.; Orchard, D.; Shewchuk, J.; Sindambiwe, E.; Storey, T.; Weerawarana, S.; Winkler, S. *Web Services Addressing (WS-Addressing)*. W3C Member Submission. http://www.w3.org/Submission/ws-addressing/

[10] BPEL4WS: Business Process Execution Language for Web Services version 1.1 2007 http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[11] Bushmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996

[12] Clark, J.; DeRose, S. XML Path Language (XPath). W3C Recommendation. http://www.w3.org/TR/xpath

[13] DILIGENT: A DIgital Library Infrastructure on Grid ENabled Technology. EC Contract No. 004260. [Online]. http://www.diligentproject.org

[14] DPM: Disk Pool Manager. [Online]. http://www.gridpp.ac.uk/wiki/DPM

[15] EGEE: Enabling Grids for E-sciencE. [Online]. http://www.eu-egee.org

[16] *eXist native XML database*. http://exist.sourceforge.net

[17] Ferraiolo, D.; Kuhn, D.R.; Chandramouli, R. Role-based Access Control. Arthec House, 2003

[18] Floros, E.; Langguth, C.; Schuldt, H.; Voicu, L. *Process Management Services Detailed Design Report*. DILIGENT Deliverable N. D1.5.3

[19] Gamma E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[20] gCube System. [Online]. http://www.gcube-system.org/

[21] GFAL: Grid File Access Library. [Online]. http://www.gridpp.ac.uk/wiki/Grid_File_Access_Library

[22] gLite Lightweigth Middleware for Grid Computing. http://glite.web.cern.ch/glite/

[23] Globus® Alliance. The Globus® Toolkit. [Online]. http://www.globus.org/toolkit/

[24] Graham, S.; Hull, D.; Murray, B. *Web Services Base Notification (WS-BaseNotification)*. OASIS Standard, October 2006

[25] Graham, S.; Treadwell, J. Web Services Resource Properties 1.2 (WS-ResourceProperties). OASIS Standard, April 2006

[26] GridSphere Portal Framework. http://www.gridsphere.org

[27] International Grid Trust Federation. http://www.igtf.net/

[28] JSR 286: Portlet Specification 2.0. http://www.jcp.org/en/jsr/detail?id=286

[29] Kropp, A., Leue, C., Thompson, R. Web Services for Remote Portlets Specification 1.0 http://www.oasis-open.org/committees/wsrp

[30] Liu, L.; Meder, S. *Web Services Base Faults 1.2 (WS-BaseFaults)*. OASIS Standard, April 2006

[31] MacKenzie, M.; Laskey, K.; McCabe, F.; Brown, P.; Metz, R. *Reference Model for Service Oriented Architecture 1.0.* OASIS Committee Specification, August 2006

[32] Maguire, T.; Snelling, D.; Banks, T. Eds. *Web Services Service Group 1.2 (WS-ServiceGroup)*. OASIS Standard, April 2006

[33] Polydoras, P.; Papanikos, G.; Kakaletris, G.; Koltsida, P.; Tsagkalidou, V.; Springmann, M.; Simeoni, F.; Aarvaag, D.; Sibeco, M. *Index and Search Services Detailed Design Report*. DILIGENT Deliverable N. D1.4.3

[34] Shoshani, A., Sim, A., and Gu, J. *Storage resource managers: essential components for the Grid*. In Grid Resource Management: State of the Art and Future Trends, J. Nabrzyski, J. M. Schopf, and J. Weglarz, Eds. Kluwer Academic Publishers, Norwell, MA, 321-340. 2004

[35] Simi, M.; Hauer, E.; F. Akal, F.; Springmann, M.; Boutsis, S.; Kakaletris, G.; Katris, D.; Koltsida, P.; Papanikos, G.; Tsagkalidou, V.; Bierig, R.; Simeoni F. *Content and Metadata Management Services Detailed Design Report*. DILIGENT Deliverable N. D1.3.3

[36] Sotomayor, B.; Childers, L. *Globus Toolkit 4: Programming Java Services*. Morgan Kaufmann, 2005

[37] Srinivasan, L.; Banks, T. Eds. *Web Services Resource Lifetime 1.2 (WS-ResourceLifetime)*. OASIS Standard, April 2006

[38] V.A. *X.509* http://en.wikipedia.org/wiki/X.509 Wikipedia

[39] V.A. *Public Key Infrastructure (PKI)* http://en.wikipedia.org/wiki/Public_key_infrastructure Wikipedia

[40] V.A. *Certification Authority* http://en.wikipedia.org/wiki/Certificate_authority

[41] V.A. *Bin packing problem* http://en.wikipedia.org/wiki/Bin_packing_problem Wikipedia

[42] V.A. *User-Community Specific Applications Detailed Design Report*. DILIGENT Deliverable N. D1.6.3

[43] Vanbenepe, W.; Graham, S.; Niblett, P. *Web Services Topics 1.3 (WS-Topics)*. OASIS Standard, October 2006

[44] VOMS http://grid-auth.infn.it/

[45] Welch, V.; Siebenlist, F.; Foster, I.; Bresnahan, J.; Czajkowski, K.; Gawor, J.; Kesselman, C.; Meder, S.; Pearlman, L.; Tuecke, S. *Security for Grid services*. In Proceedings of 12th IEEE International Symposium on High Performance Distributed Computing, pages 48-57, June 2003

[46] Wiederhold, G., Wegner, P., Ceri, S. *Towards megaprogramming*. Communications of the ACM 35(11), 89-99 (1992)

[47] Apache Lucene (http://lucene.apache.org/)

[48] GeoTools, The Open Source Java GIS Toolkit (http://geotools.codehaus.org/)

[49] JDBM (http://jdbm.sourceforge.net/)