Grant agreement no. 211714

neuGRID

A GRID-BASED e-INFRASTRUCTURE FOR DATA ARCHIVING/ COMMUNICATION AND COMPUTATIONALLY INTENSIVE APPLICATIONS IN THE MEDICAL SCIENCES

Combination of Collaborative Project and Coordination and Support Action

Objective INFRA-2007-1.2.2 - Deployment of e-Infrastructures for scientific communities

Deliverable reference number and title: D6.2 Interim service prototype report

Due date of deliverable: Month 24

Actual submission date: 31st January 2010

Start date of project: February 1st 2008 Duration: 36 months

Organisation name of lead contractor for this deliverable: **P3** University of the West of England, Bristol UK

Revision: Version 1.0 – First Draft release for Review.

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Table of Contents

Purpose and Intended Audience of this Document 3
Executive Summary.....4
1. The Pipeline Service.....7
2. The Provenance Service 17
3. The Glueing Service.....27
4. The Querying Service.....35
5. The Portal Service 40
6. The Anonymisation Service 47
7. The Way Forward.....53

Document Revision History

What	To Whom	Comments	Subjects involved	Dates
1st Draft	Whole consortium	Author to circulate the 1st draft to the whole Consortium	AA	1/1/10
1st Draft + comments	Author	1st review: Reviews due back to the Author. Author to implement comments	Alex + David	1/10/10
2nd draft	PMT	Author to circulate the 2nd draft to PMT, Technical Supervisor and Area leaders	AA	1/13/10
2nd draft + comments	Author	2nd review: PMT's reviews due back to the Author. Author to implement comments	PMT	1/17/10
Final Draft	PC	Author to send Final draft to the Project Coordinator	AA	1/25/10
Final Draft + comments	Author	Coordinator's review due back to the Author	GbF	1/27/10
Deliverable	CB	Coordinator's review - Author sends final draft to GbF	AA	1/29/10
Deliverable	Commission	Submission - Final changes made and submitted to the Commission	CB	1/30/10

Purpose and Intended Audience of this Document

D6.2 The interim service prototype report forms a public deliverable and documents the progress achieved on the services implementation in WP6. In this workpackage (WP), a set of generalised services is being designed and implemented that will help the users in their analysis. WP leaders, technical users, neuGRID developers, project managers and EC reviewers are the intended recipients of this document which may be revisited as and when key recommendations presented in the document evolve, due to the ongoing research and development process in the workpackage. To a lesser extent, since the recommendations and future plans in the workpackage may impact the decisions made on the potential use and exploitation of the project outcomes, neuro-scientists and prospective users (e.g. Pharmaceutical industries) as well as internal and external users of the project activities, are also anticipated as potential readers of this deliverable.

Executive Summary

The aim of the neuGRID Project is to provide a user-friendly grid-based e-infrastructure plus a set of generalised infrastructure services that will enable the European neuroscience community to carry out research that is necessary for the study of degenerative brain diseases. The WP6 work-package, the provision of Distributed Medical Services, is responsible for supplying general purpose analysis services to the users of the project. The provision of these generalised medical services will enable grid technologies to be applied in this and a number of other medical domains. The services will provide the flexibility that is necessary for interfacing with existing medical systems and will enable the reuse of packaged services that exploit grid functionality. This work package will gradually provide an Application Programming Interface (API) which is independent both of the application domain and of the underlying Grid infrastructure.

In order to achieve these objectives, a requirements analysis process was carried out to identify the group of services that are suitable for addressing the neuGRID project objectives (as reported in D9.2 from WP9). The requirements process also helped in identifying the functionality that these services should provide for the users of the system. During this activity a design philosophy to drive the services design process was produced. The design philosophy delivered a set of guidelines that services should follow to ensure their execution and eventual composition into biomedical applications. The WP6 deliverable document in year 1 outlined the design philosophy that is being followed during the construction of the distributed medical services. The deliverable also described a design for the set of services that will constitute the distributed medical services layer. The design and evaluation process was led by user requirements, which have been separately elicited in WP9 deliverables.

Once the project requirements had been elaborated, the next step was to map these requirements against possible components and identify those that should be used. The reasons for building these components as services have been presented in detail in the design philosophy section in year 1 deliverable D6.1. The services will exist as autonomous and loosely coupled entities that can be executed independently. Each service will address requirements that cannot be handled by any other single component or service. Collectively and in cooperation with each other, services will support the user analysis process and therefore deliver a functional neuGRID system.

A group of services has been identified that is neither middleware nor application specific. These generalised services can be used by any application and should run on any grid middleware. We have discussed the major components and requirements that each of these generalised services should address as well as a more detailed design for each of these services in the year 1 deliverable D6.1. This also included architectural considerations and the selection of the most suitable structure for each of the services. The services design also considered the technological choices which were available and justified why specific technologies have been selected. The design included individual service components, API's and interfaces that will be provided to enable interaction with other services and applications.

In year 2, efforts were made to finalise the designs of the services whose initial specifications were submitted in the year 1 deliverable and that can best address the user requirements. At the same time, it was ensured that the service designs are consistent with the design principles that were presented in the year 1 deliverable. A

significant effort has been invested in year 2 to implement the services. Each of these services has an implementation roadmap that closely aligns with project commitments and deliverable submission deadlines. The progress achieved on the service implementation is reported in this year 2 deliverable. The following set of activities was performed in year 2 on the design and implementation fronts:

- The design of each service was finalised. This design process not only glued together various components, it also ensured that these components and services can be extended as and when required. It was also ensured that minimum dependencies existed between components and services and that they were sufficiently scalable to cope with future demands in loads. Particular care was taken to make these services as generalised as possible and vendor and middleware lock-ins were avoided.
- Suitable interfaces were crafted to provide access to these services. It was ensured that these interfaces promoted interoperability and ease-of-use. Standard approaches were employed in designing these interfaces with the intention that services should follow generally agreed standards to the greatest possible extent in order to make them widely exploitable.
- This deliverable document highlights the progress that has been achieved on each of the services. This document also touches upon the important features that each of the services offers as well as technical limitations that were faced in implementing the proposed services designs. The service descriptions also include the work that will be carried out in the third year of the project. The details have also been provided regarding the use of interfaces, technologies and about the implementation as well as the deployment environment used in these services. Any missing functionality is described and a roadmap for implementing and delivering the remaining functionality has also been set out.

A summary on the status of the services is presented in the following section. The detailed services reports can be found in the respective sections in this document.

a. *Pipeline Service:* The role of the Pipeline Service is to enable the users to create and design workflows in a user-friendly fashion in a workflow authoring environment of their choice. The current implementation of the Pipeline Service consists of the following implemented features: 1) A fully defined web service interface of the Pipeline Service; 2) An implemented translation component that supports translations both for the LONI Pipeline, and translation to JDL for gLite Submission; 3) An enactor that uses an extended gLite-adaptor for the Glueing Service for submission to the grid.

b. *Glueing Service:* All the generalised services in neuGRID will access distributed resources through a Glueing service. This service will provide an abstraction layer through which users can access data and other resources without lock-in to a particular middleware. The current implementation of the service encapsulates the following functionality: 1) Job Submission; 2) File Management (reading, writing, directory listing, getting file size); 3) Job Monitoring; 4) User Authentication and 5) LORIS Integration.

c. *Provenance Service:* The Provenance Service will capture, store and perform analysis on the data for improved decision making. The following features are available as a consequence of the CRISTAL adoption as the provenance management tool: 1) Provenance capturing from pipelines; 2)

Provenance data storage in files; 3) Workflow life cycle management and 4) Mechanism to store user specific provenance.

d. *Querying Service:* The Querying Service will be a generalised service that can query and browse data that is stored in a file or relational database or in other Grid databases. Due to the dependence on provenance service and LORIS API's, the querying service is not yet available for use, however, the following progress has been made: 1) The service design is finalized; 2) Technology evaluations are complete and 3) Interfaces have been defined.

e. *Portal Service:* The Portal Service will be a point of entry to the system and all other services will be accessed through this service. This service will also hide the low-level service interfaces and implementation details from common users. The following components are available in the portal service: 1) The Single Sign-On (SSO) system based on the Central Authentication Server (CAS); 2) The dashboard (menu) and 3) The neuGRID JSR286/WSRP 2.0 compliant portal.

f. *Anonymisation Service:* Biomedical data needs to be anonymised and should be shared after ethical and legal clearance. The anonymisation service will ensure this before the users can access biomedical data for their analysis. The following components are available in the anonymisation service: 1) The pseudonymisation library; 2) The Pseudonymisation Web Service and 3.) The Pseudonymisation applet/stand alone application.

The deliverable document concludes with a roadmap for the third year which is presented in the "way forward" section.

As discussed in the design document, user requirements may evolve and new sets of features may need to be added. Consequently, service designs and the functionality offered may also evolve and provision has been made in the designs to address potential future changes in functionality. The design and implementation will therefore be periodically reviewed and any changes and future suggestions and recommendations will be considered. It is anticipated that most of the essential requirements that have been identified in the user requirements analysis will be implemented before the project concludes by the end of year 3. Service libraries as well as documentation will be released as and when available according to the project plan and will be integrated with the rest of the project deliverables. The services will also be demonstrated to the potential user communities and their feedback will be used as a vehicle for service testing, improvements and production quality releases.

1. The Pipeline Service

1.1 Purpose and Introduction

The neuGRID generalised medical services layer includes numerous components that facilitate the execution of a neuroimaging pipeline on a grid infrastructure. One of the central services enabling this is the Pipeline Service. The functionality of the Pipeline Service is mandated by specific requirements from WP9. The role of the Pipeline Service is to enable scientists to create and design workflows in a user-friendly fashion in any workflow authoring environment of their choice. The Pipeline Service will also plan and distribute the pipeline over a grid, and finally coordinate with the Provenance Service to enable users to retrieve and query the results of the execution. The purpose of this section is to update the stakeholders about the implementation progress of the Pipeline Service and highlight issues and current limitations that will be addressed in future versions.

1.2 Architecture

The components of the Pipeline Service are outlined in Figure 1. The interaction starts with the authoring of a pipeline, which the user wants to execute on the Grid. Authoring can be done in numerous tools. LONI Pipeline is a popular neuroimaging pipeline authoring environment that is supported by the Pipeline Service. The Pipeline Service implements flexible interfaces for integrating any suitable authoring environment (described in more detail in section 3.2).

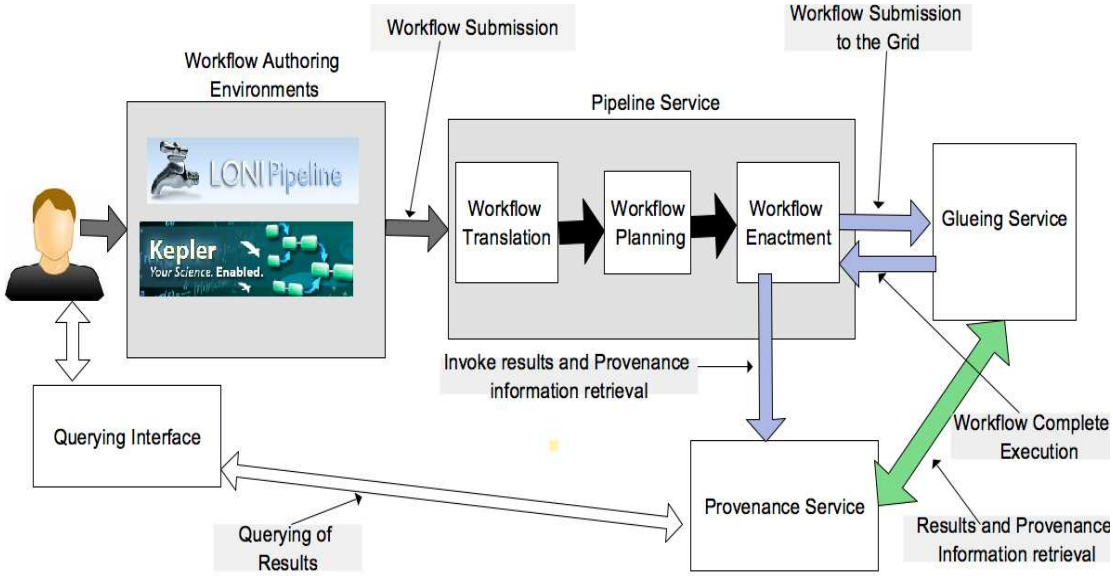


Figure 1: Pipeline Service Architecture

After authoring the pipeline, the user invokes its submission to the Pipeline Service. Upon submission several things occur: first the authored pipeline that is represented in a native format to the authoring environment is translated into a common objected-oriented workflow model. A draft specification of this model was presented in the year 1 deliverable document D6.1. This representation has been expanded to accommodate further use cases and support other workflow paradigms such as service-based workflows. A full specification of the object-oriented workflow language is presented in section 3.2. After workflow translation, the workflow is planned for the execution. In workflow planning the abstract user specified workflow

is transformed into a concrete executable workflow. The workflow structure may also be modified for more efficient execution in the grid.

Workflow enactment is the final stage in the Pipeline Service. The Pipeline Service submits the planned workflow to the Grid via the Glueing Service. The Glueing Service abstracts all interactions between WP6 Services and the underlying Grid middleware. In the case of the Pipeline Service, the functionality that enables submission of a workflow to the grid and workflow monitoring capabilities are used primarily. Once a workflow has been executed, the Pipeline Service invokes the Provenance Service to initiate the retrieval of the provenance information and the output of the workflow. The user can then query the results through the Querying Service.

1.3 Current Implementation

The current implementation of the Pipeline Service consists of the following implemented features.

- A fully defined webservice interface of the Pipeline Service.
- Implemented translation component that supports translations for both LONI Pipeline, and translation to JDL for gLite Submission.
- Enactor that uses an extended gLite-adaptor for the Glueing Service for submission to the grid.

Each implemented feature will be described in detail in the subsequent sections.

1.3.1 Web Service Interface

According to the WP6 design philosophy the Pipeline Service has been designed around a Service Oriented Architecture (SOA). A preliminary draft Web Service interface has been presented in the Year 1 WP6 design deliverable D6.1. This interface has been enhanced in response to the feedback from project partners. The webservice binding enables clients to interact with the Pipeline Service and perform various functions such as submission of workflows, tracking progress and various control functions. There are essentially four types of methods supported in the Pipeline Service. Submission methods expose functionality that enables a user to submit a workflow. Workflow Control methods expose functionality that enable interaction with currently executing workflows. The Pipeline Service Initialisation method is invoked to initialise a session with the Pipeline Service. Finally the Pipeline Service Internal methods are methods that enable interaction with internal components of the Pipeline Service. Specific parameters are required for various Pipeline Service methods. The following four parameters are generally used*: SessionID, Workflow, UserCredentials* and *WorkflowID*. The *SessionID* denotes the identifier used internally by the Pipeline Service to identity a specific user session. The *Workflow* argument denotes the actual workflow submitted by the user. The *userCredentials* argument denotes a credential provided by the user to identify the user. The *WorkflowID* argument used in the Workflow Control methods is the middleware-specific unique identifier for the workflow.

The Pipeline Service API methods are as follows. Details of each method are provided in subsequent sections.

Submission Methods

- *batch_submit(SessionID, Workflow)*
- *int_submit(SessionID, Workflow)*

Workflow Control Methods

- *cancelWorkflow(SessionID, userCredentials), cancelWorkflow(WorkflowID)*
- *getStatus(SessionID, userCredentials), getStatus(WorkflowID)*
- *getWorkflowOutput(SessionID,userCredentials), getWorkflowOutput(WorflowID)*

Pipeline Service Initialization Method

- *initSession(UserCredentials)*

Pipeline Service Internal Methods

- *getWorkflow(SessionID, userCredentials), getWorkflow(SessionID)*
- *translateWorkflow(workflowType, sessionID)*
- *planWorkflow(planner, Workflow, sessionID)*
- *submitToGS(workflow, sessionID)*

1.3.1.1 Submission Methods

Two submission functions are provided as part of the Pipeline Service. One submit function, *batch_submit,* is provided for submission of workflows which are not interactively monitored. While *int_submit* is provided for interactively monitored workflows, such as those created from an authoring environment e.g. the LONI Pipeline. Both functions take a *sessionID,* which maps a particular user to the submitted workflow and a workflow specification is provided as a SOAP attachment.

1. batch_submit(SessionID, Workflow) Method
The *batch_submit* function is provided for submitting workflows which are compute and data intensive. These workflows will take a considerable amount of time to execute, hence users may not track the progress of the workflow interactively. To optimize the execution of these compute and data intensive workflows the Pipeline Service will use the workflow planner. The Pipeline Service will also manage the transfer of the output and logging information of the workflow to the Provenance store for later querying and analysis.

The sequence chart is shown in Figure 2.

Steps 1-3: Users first create a new session which uniquely identifies the workflow the user is submitting. A new session is created by the *initSession(UserCredentials)* function. The *UserCredentials* argument is provided by the User and will map a specific user to the submitted workflow. *UserCredentials* will also be the primary means of authentication in order to authorize use of the Pipeline Service.

Steps 4-6: The PS Controller is a component of the Pipeline Service that will orchestrate all activities required on behalf of users. The PS Controller will export the Pipeline Service API via a webservice binding. The Pipeline Service will support numerous workflow formats. The Translation Component is provided in order to convert supported formats into a common standard format. The components within the Pipeline Service will operate on the standard format.

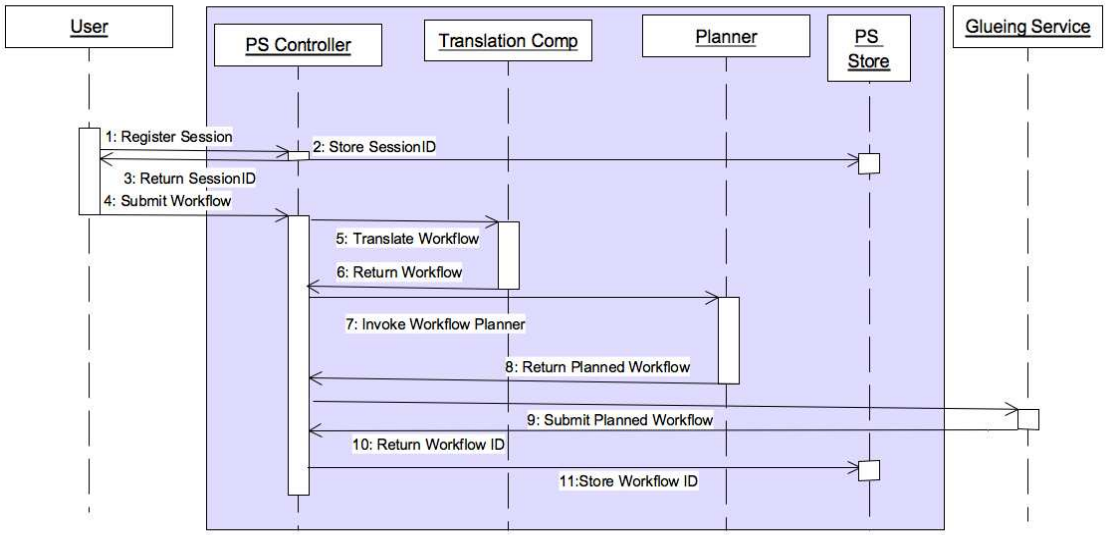


Figure 2: Sequence Diagram for *submit* Function

Steps 7-8: After the workflow has been converted into a common format, the PS controller forwards the workflow to the Planner for optimization.

Steps 9-10: After receiving the planned workflow from the Planner, the PS Controller submits the workflow to the Glueing Service via an embedded Glueing Service client. The need for a planner and the role it will play in optimizing the workflows has been discussed in D6.1. The Glueing Service will take the workflow and through appropriate adaptors convert it into a middleware specific format e.g. JDL in gLite. The Glueing Service will return a middleware specific workflow identifier which will be used to track the progress of the workflow and retrieve output upon completion.

2. int_submit(SessionID, Workflow) Method

The *int_submit* function is provided for submitting workflows though GUI authoring and submission environments such as the LONI Pipeline. GUI authoring and submission environments provide mechanisms for users to author workflows, to submit them and to track progress interactively. Due to continuous monitoring of the workflow there are some differences in the functionality of this method compared to *batch_submit*. GUI environments will be primarily used to create new workflows or customize existing ones and execute them to determine their behaviour. GUI environments will also be used to debug existing workflows. The workflow author will determine the workflow correctness by checking the output of each stage of the workflow execution, and in case of errors check the logs to determine errors. For this reason the workflow planner is not used in the *int_submit* function. The Workflow Planner, due to its optimization strategies, may change the workflow structure and may eliminate the execution of certain tasks based on data availability. This optimization may disrupt the workflow authoring process. To author a new workflow or correct existing ones, the workflow that is executed on a Grid, should be identical to the workflow that was authored. Once the workflow is deemed to be complete and frozen, then workflow optimization can be applied through the *submit* function.

The sequence chart is shown in Figure 3.

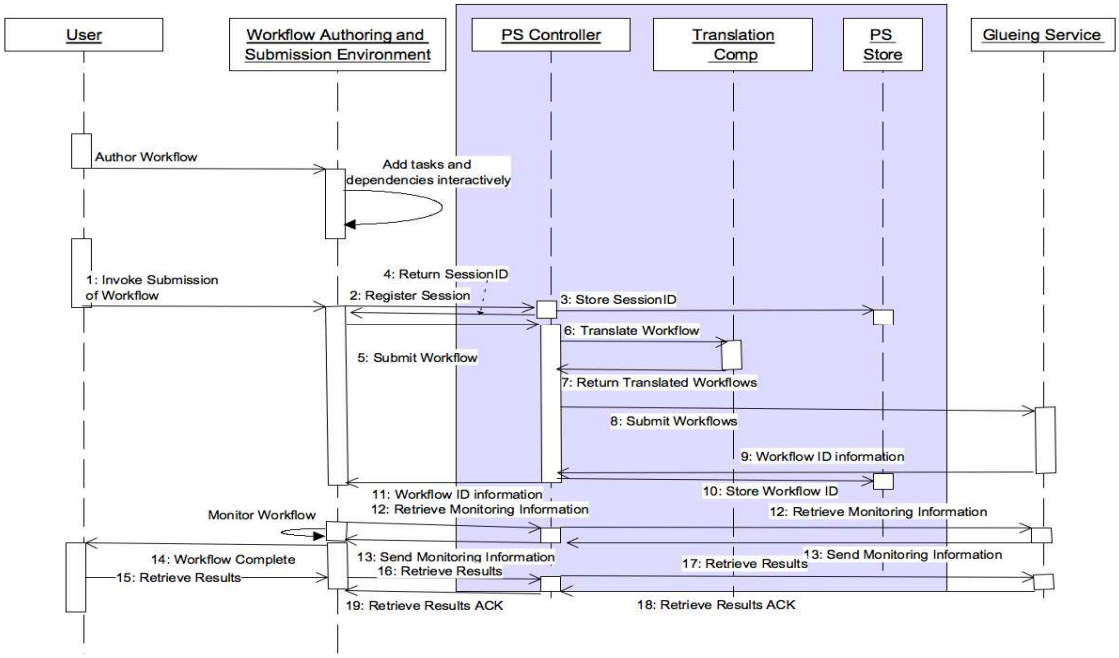


Figure 3: Sequence Diagram from *int_submit* Function

Steps 1-4: Users first author a workflow in the environment and invoke submission. The Workflow authoring environment will first register a new *sessionID* from the Pipeline Service.

Steps 5-11: The authoring environment submits the workflow to the Pipeline Service with the *int_submit* function. The Pipeline Service at first translates the workflow definition into a common format and then submits the workflow without planning to the Glueing Service. The Glueing Service forwards it to the appropriate middleware adaptor and returns a middleware specific workflow identifier. This identifier is stored within the Pipeline Service and returned to the authoring environment.

Steps 12-14: The authoring environment tracks the progress of the workflow through the *getStatus(WorkflowID)* method. The *getStatus* method will invoke the appropriate SAGA monitoring functions, which will in turn invoke the middleware specific job status mechanism e.g. gLite-wms-job-status and gLite-wms-job-get-logging-info in gLite. Once the job is complete the authoring environment will quit monitoring.

Step 15-19: Once the job is complete, users can specify a server to which the output of the workflow is to be transferred. This request is forwarded to the Glueing Service which then uses middleware specific mechanism to transfer workflow output e.g. gLite-wms-job-output in gLite.

1.3.1.2 Workflow Control Methods

The Pipeline Service provides numerous workflow control methods which enable the following.

1. *Cancel a workflow*
2. *Get status of a running workflow*
3. *Retrieve output of a workflow*

One thing to note about these functions is that two variants of the methods are provided. One class of functions takes the *workflowID* as argument, while the other takes both the *sessionID* and the *userCredentials* as argument. The *(sessionID, userCredentials)* functions are designed to be used when users use the Pipeline Service in multiple sessions. For instance users can submit workflows through the *batch_submit* function, and after some time may want to determine the status of a workflow and retrieve the output once it has been completed. In this scenario the users will provide *sessionID* of the workflow session and *userCredentials* argument. The Pipeline Service will retrieve the *workflowID* form the PS store and contact the Glueing Service to retrieve the appropriate information. The *workflowID* functions are designed to be used in a single continuous session.

1.3.1.3 Pipeline Service Initialization Method

The *initSession* method is the basic method used to initiate a new session with the Pipeline Service. The identifier returned is a unique ID.

1.3.1.4 Pipeline Service Internal Methods

The Pipeline Service Internal Functions are a set of functions which are used internally by the Pipeline Service, but an external webservice binding is provided in order to give the Pipeline Service an open architecture. For instance *translateWorkflow*, *planWorkflow* and *submitToGS* are functions which are used by the PS controller to orchestrate the workflow translation, planning and submission of a Workflow respectively. These functions are exposed to enable simplified and open usage of the Pipeline Service. For instance, if a user has an appropriately planned workflow there is no need for the user to use the batch or interactive submit functions rather the user can directly call the *submitToGS* function to submit the workflow directly to the Glueing Service. Similarly if the user wants to determine how a workflow will be planned, the user can call the *planWorkflow* function and review the planned workflow.

Another design consideration for exposing these functions is that developers can customize the way they use the Pipeline Service. A developer can create his own PS client which invokes the Translation Component but uses a third party workflow planner and then submits the output of the planner to the Glueing Service.

1.3.2 Translation Component

The Pipeline Service is designed to support multiple workflow specification formats. For this purpose an object-oriented workflow API has been designed. The objective of the API is to enable the translation of the most common workflow formats to a common format which the Pipeline Service components can interact with. A draft of the API was presented in Year 1 deliverable document D6.1. However in response to partner feedback the API has been extended to support Web Service workflows as well. The Web Service workflow extension as of yet is incomplete and the task API is being developed and implemented. This section proceeds as follows. The workflow API is depicted in Figure 4 and described subsequently. Dynamic instantiation of appropriate translators during the runtime of the Pipeline Service is essential to support such an API. Hence the mechanism used in the Pipeline Service is described in section 3.2.2. This section also describes how new translators can be created for the Pipeline Service.

1.3.2.1 Objected Oriented Workflow Representation

The Translation component implements an API which allows the translation of various workflow specification formats to a common format. The component will be designed to convert a workflow description to a common format used within the Pipeline Service. The following is a description of the classes of the Translation component. The class diagram is shown in Figure 6.

1. Activity Abstract Class

This class represents all the entities that are manipulated inside a Workflow.

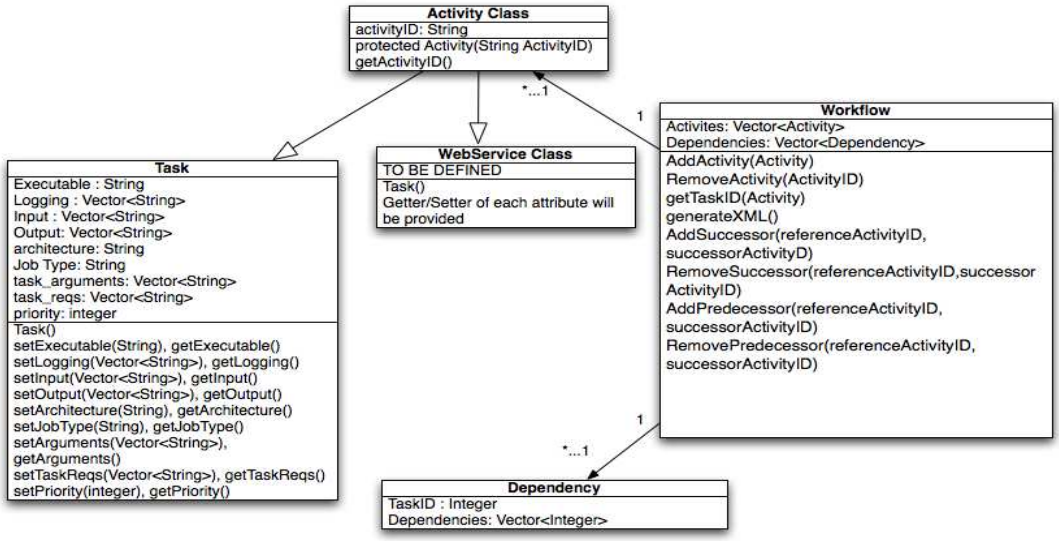


Figure 4: Class Diagram of the Translation API

2.WebService Class (Extends Activity)

This Class contains everything that is needed to represent a WebService

3. Task Class (Extends Activity)

The Task Class contains properties that define an executable task in a Workflow. It contains properties that represent Executable Information, Logging Information, Input Data information, Output Data Information, Architecture specific information, Job Type, Task Arguments, OS environment, Task Requirements, and Priority of the job.

It is important to note that not all workflow specifications record all of these properties. However, for LONI Workflows, some properties are recorded that are not represented in this class. For this the Task class can be inherited and extended in a sub class. This has been done for the LONI Adaptor.

4. Workflow Class

The Workflow class is the class which defines the structure of the workflow representation within the Translation component. The Workflow class contains properties that enable the declaration of *Dependencies and the activities in the workflow*.

5. Dependency Class

The Dependency class defines a structure for declaring a dependency. The basic properties in this class include an ActivityID property which defines the ID of the task

concerned. The other property is the dependencies property which is a vector of ActivityID, which enumerates all IDs of activities which have a dependency relationship with the activity.

1.3.2.2 Dynamic Instantiation of Translators

Once a workflow has been submitted to the Pipeline Service, the *PipelineController* is instantiated to manage the entire life-cycle of the workflow within the Pipeline Service. As depicted in figure 1, the first stage of a submitted workflow is the workflow translation. As can be seen in Figure 5, the Translator is invoked by calling the method *translate* with parameters *sessionID* and *workflowType*. As previously mentioned the *sessionID* identifies the user's session. The *workflowType* identifies the format of the workflow. The Pipeline Service maintains an internal storage where it stores users sessions and the corresponding workflows submitted. The internal storage is also used to store translated workflows as well as the XML marshalled object-oriented representation of a workflow. In the UML activity diagrams presented in figure 2 and 3, this storage is termed as *PS Store*. As can be seen in figure 5, the *PSStoreWrapper* class encapsulates interaction with the storage in the current implementation.

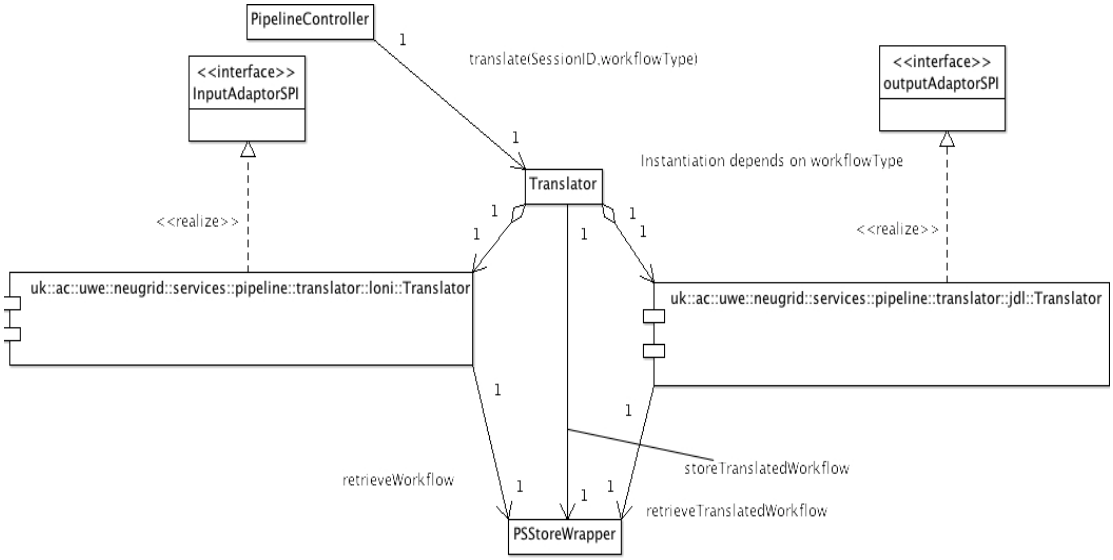


Figure 5: Class Diagram of the Translator

As depicted in figure 5, the translator dynamically instantiates an appropriate translator based on the workflow type. To enable dynamic instantiation of an appropriate translator, specific interfaces have been designed to format specific workflows. Each adaptor implements a specific Service Provider Interface (SPI). The *Translator* package provides two adaptors interfaces *InputAdaptorSPI* and *OutputAdaptorSPI*. The *InputAdaptorSPI* defines the interfaces that an adaptor must follow to support translation from a native workflow format to the Pipeline Service object-oriented workflow format. *OutputAdaptorSPI* defines the interfaces that must be implemented to provide translation from the workflow object-oriented format to a specific format for submission.

Besides following a specific protocol, the translation package must be registered with the Pipeline Service. This is currently handled by providing a line identifying a Translator to a specific package. Once a translator is registered with the Pipeline

Service and follows the appropriate interface, it can be dynamically instantiated by the Pipeline Service.

1.3.2.3 Process for adding a new translator

There are two steps required to create a translator for the Pipeline Service.

1. Implement the Input/OutputAdaptorSPI

The InputAdaptorSPI and OutputAdaptorSPI provide interfaces that need to be implemented to convert a workflow from a native format to the Object Oriented format (in case of InputAdaptorSPI) and from an object oriented format to a format for submission (in case of OutputAdaptorSPI).

2. The translator must be registered with the Pipeline Service. The Pipeline service uses the PSConfig.xml file to look for available translators.

1.3.3 Enactor

The current implementation of the Enactor submits the JDL to a specifically extended adaptor for the gLite middleware. The Pipeline Service enactor is responsible for the submission of a workflow to the grid infrastructure. The workflow received by the enactor is a concretely planned and transformed. Currently workflow planning is not implemented in the Pipeline Service and is a future task. As depicted in figure 6, the enactor uses a SAGA-based client to invoke the UWESOAPAdaptor. All interactions with the Glueing Service are abstracted through the UWESOAPAdaptor at the Pipeline Service Side. The UWESOAPAdaptor forwards translated SAGA requests in a SOAP format and invokes the Glueing Service. The Glueing Service uses the gLite-adaptor to submit workflows to a gLite-based Grid infrastructure. The gLite-adaptor initiates a user proxy and submits the workflow to glite-WMS.

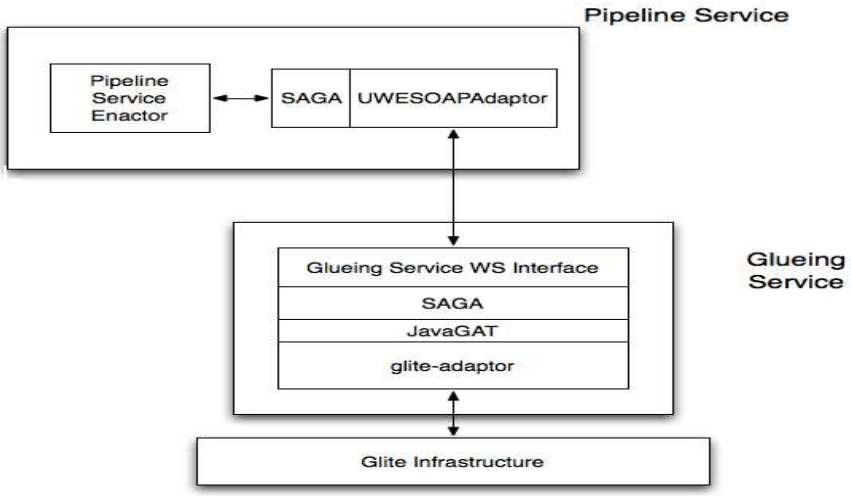


Figure 6: Enactor Architecture

To facilitate this process, the Pipeline Service provides the following parameters to the adaptor, however, once the system is integrated with the system-wide single-sign on, this should not be required. This is mentioned in the subsequent section titled “Integration with Single Sign On”.

1. Grid Infrastructure Specific: VO Name, VO Host DN, VOMS Server URL, VOMS Server Port

2. User specific: User Certificate, User Key, User Passphrase

1.4 Issues and Limitations

The current issues facing the Pipeline Service are highlighted in this section.

1. Glueing Service Specific Considerations

The Pipeline Service needs to coordinate the results retrieval with the Glueing Service and the Provenance Service. Current monitoring information received from the Glueing Service is inadequate for comprehensive Provenance. The Glueing Service needs to be extended to gather scheduling information, detailed logs of tasks in addition to the output specified in the JDL. Additionally to support interactive monitoring of tasks in the Pipeline Service, the Glueing Service needs to be extended to enable monitoring of individual tasks in a workflow.

2. Integration with Single Sign On

Currently to invoke enactment of a workflow, the user's certificate, key and associated passphrase are required to initiate a proxy at the Glueing Service end. To use this model the user has to provide all of these details every time a submission is invoked. To cater for this limitation the Pipeline Service needs to be integrated with a Single Sign On service.

3. Integration with the Grid Information Services

Currently workflow planning is not implemented, however for efficient planning knowledge of the Grid environment is required. The information, such as how many sites are available, which replicas are present and where different tasks have been deployed, is required to efficiently plan a workflow. The Glueing Service needs to get this information from the Grid information services. Currently such functionality is not present in SAGA.

1.5 Future directions

A major component of the Pipeline Service that has yet to be implemented is the workflow planner. The issues raised in the previous sections directly have an impact on the workflow planner. Hence in year 3, the issues that have been raised will be addressed and the workflow planner will be implemented to complete the proposed Pipeline Service architecture.

2. The Provenance Service

2.1 Introduction

A scientific workflow is a formal specification of a scientific process, which represents, streamlines, and automates the steps from dataset selection and integration, computation and analysis, to final data product presentation and visualization. A workflow management system supports the specification, execution, re-run, and monitoring of scientific processes. Such workflow processes have a level of complexity that may lead to human error, which cumulatively have a large impact on the validity of the results that are produced. Researchers therefore require a means of tracking the execution of given workflows so they can ensure that important results are accurate. Currently this is carried out manually before research is released to the wider community and is published.

The neuGRID provenance service is primarily intended to capture and provide the information that is necessary during this process. The provenance service will keep track of the origins of the data and its evolution between different stages and services. Provenance metadata captures the derivation history of a data product, including the original data sources, intermediate data products, and the steps that were applied to produce the data product. The provenance service will allow users to query analysis information, to regenerate analysis workflows, to detect errors and unusual behaviours in past analyses and to validate analyses. The service will support and enable the continuous fine-tuning and refinement of the pipelines in the neuGRID project by capturing:

1. Workflow specifications.
2. Data or inputs supplied to each workflow component.
3. Annotations added to the workflow and individual workflow components.
4. Links and dependencies between workflow components.
5. Execution errors generated during analysis.
6. Output produced by the workflow and each workflow component.

In the past few years UWE has been working with partners from CERN and CNRS, France to develop a data and workflow tracking (i.e. provenance) system entitled CRISTAL which is now being used to track the construction of large-scale experiments at the CERN Large Hadron Collider (LHC). The M1i company is currently working with UWE to transfer CRISTAL technology to regional French companies under the product name of Agilium, for the purpose of supporting business process management (BPM) and the integration and co-operation of multiple processes especially in business-to-business applications. In essence CRISTAL/Agilium is being developed as a business process modelling and provenance capture tool. The product addresses the harmonisation of business processes by the use of the CRISTAL kernel so that multiple potentially heterogeneous processes can be integrated with each other and have their workflows tracked in the database. Using the facilities for description and dynamic modification in CRISTAL in a generalised and reusable manner, Agilium is able to provide modifiable and reconfigurable business process workflows. It uses the so-called description-driven nature of the CRISTAL models to act dynamically on process instances already running and can thus intervene in the actual process instances during execution. These processes can be dynamically (re-)configured based on the context of execution without compiling, stopping or starting the process and the user can

make modifications directly and graphically of any process parameter, while preserving all historical versions so they can run alongside the new. In the provenance service, thorough investigations have helped us to use CRISTAL to provide the provenance needed to support neuroscience analyses and to track individualised analysis definitions and usage patterns thereby creating a knowledge base for neuroscience researchers.

2.2 CRISTAL Based Provenance Service Architecture

This section describes how CRISTAL will fit into the overall neuGRID architecture to capture and coordinate provenance data. The subsequent sections explain the implementation details of CRISTAL and highlight how provenance is captured, modelled, stored and tracked through the course of an analysis.

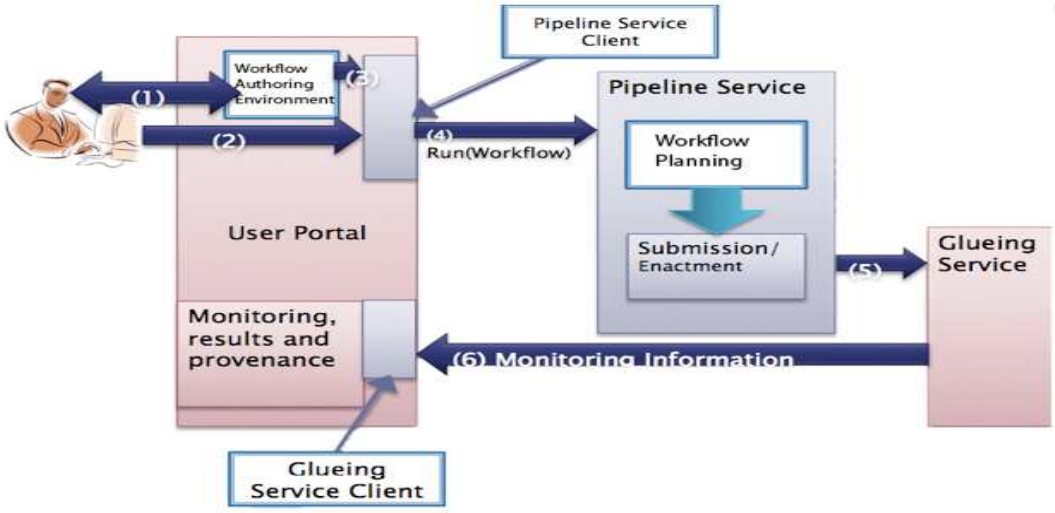


Figure 7: The neuGRID Services without provenance support

As shown in figure 7, the interaction starts with the authoring of a pipeline (workflows are called pipelines in the context of the neuGRID project), which the user wants to execute on the Grid (1). Authoring can be carried out via several tools, the prototype being implemented in neuGRID, uses Kepler and the LONI Pipeline as examples of authoring environments. The pipeline architecture is flexible and any suitable authoring environment can be accommodated. After authoring the pipeline, the user invokes the submission of the pipeline (2). In this case, several things happen: first the authored pipeline, which is represented in a Modelling Markup Language (MoML) format (in the case of Kepler) or in LONI Pipeline XML (in the case of LONI) is transformed into a simple XML based workflow format, which is passed to the Pipeline Service (3). This then translates the specification into a workflow object, via an API, which will be provided as part of the Pipeline Service. The workflow object is translated into a DAX file, via the Pegasus DAX API. Pegasus is used as a workflow-planning tool in this environment. The DAX file represents the abstract workflow that the user has defined. Using the resources information that is available in a distributed infrastructure (in neuGRID's case, a glite based infrastructure) Pegasus plans the workflow into a concrete executable workflow. The following operations are carried out by Pegasus on the workflow:

- Tasks are mapped to individual computing resources, depending on availability of task actors and/or study set replicas or partial workflow outputs.
- Portions of the workflow are mapped to specific computing resources, depending on the computing platforms and computing resources provided by the sites.
- The workflow specification is enhanced by including data staging actors to stage data between computational sites.
- The workflow specification is enriched by including provenance actors for provenance collection.

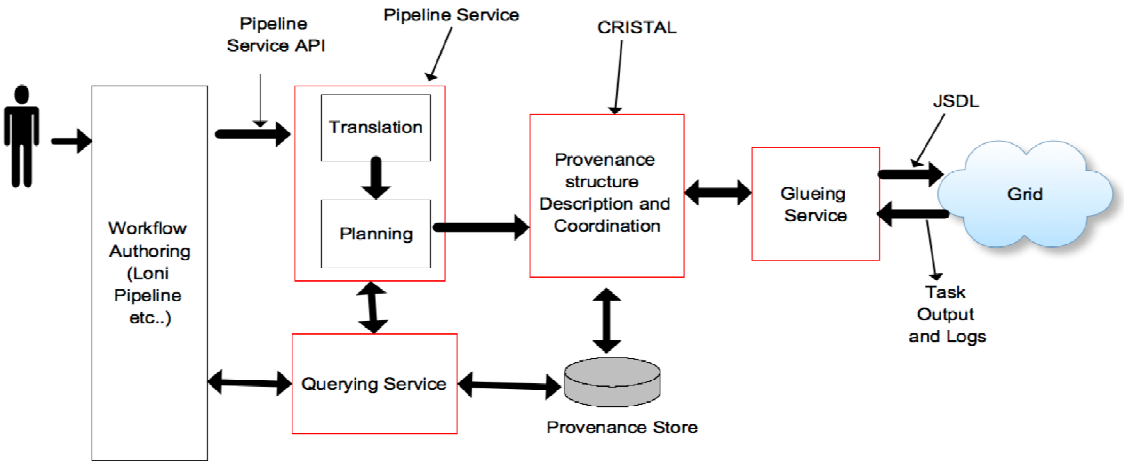


Figure 8: The neuGRID Services with provenance support

The Pipeline Service translates the workflow specification into a standard format and plans the workflow. The planned workflow, as shown in figure 8, is forwarded to the CRISTAL enabled provenance service which then creates an internal representation of this workflow and stores the workflow specification into its schema. This schema has sufficient information to track the workflow during subsequent phases of a workflow execution. The workflow activity is represented as a tree like structure and all associated dependencies, parameters, and environment details are represented in this tree. The schema also provides support to track the workflow evolution and the descriptions of derived workflows and its constituent parts are related to the original workflow activity.

The provenance service provides a provenance-aware workflow instantiation engine. The workflow is broken into its constituent jobs and CRISTAL takes care of the jobs, their dependencies and the order in which these should be executed to complete a workflow. CRISTAL coordinates the whole job execution process and the jobs wait inside the CRISTAL premises if their dependent tasks are in execution. The workflow is instantiated in a task-by-task manner by CRISTAL. These tasks are sent to the Glueing Service for execution in the Grid and the results and logs are retrieved to populate a provenance structure. CRISTAL is unaware of how the actual scheduling, task allocation and execution is carried out in the underlying Grid infrastructure. All

of these operations are performed independently from CRISTAL. The information stored in the provenance structure can be interactively queried by users.

2.3 Workflow Instantiation and Execution in CRISTAL

Figure 9 shows the proposed integration of CRISTAL with the neuGRID services architecture in greater detail. The Pipeline Service will forward a concrete planned Workflow in an *XML format* to CRISTAL. The CRISTAL wrapper is the first component that will receive this workflow from the pipeline service and will perform a number of actions on the workflow. In neuGRID, the wrapper will be a webservice and will accept SOAP calls to allow compatibility and interoperability with other neuGRID services. The wrapper will process the workflow that has been received as a SOAP request and will populate an internal CRISTAL structure from this workflow. The Provenance service will coordinate with the pipeline service to receive a workflow and store the captured provenance for further tracking and analysis. Once the workflow has been instantiated and populated in the CRISTAL structure, CRISTAL will coordinate with the Glueing service to submit the workflow as a whole or in parts/tasks for ultimate scheduling and execution in a Grid environment. When a workflow or one of its tasks has been executed, the execution as well as the state logs will be sent back to CRISTAL for provenance storage and management. CRISTAL will also take care of the dependencies between various tasks in a workflow and organize the information that is captured during the instantiation and execution phases.

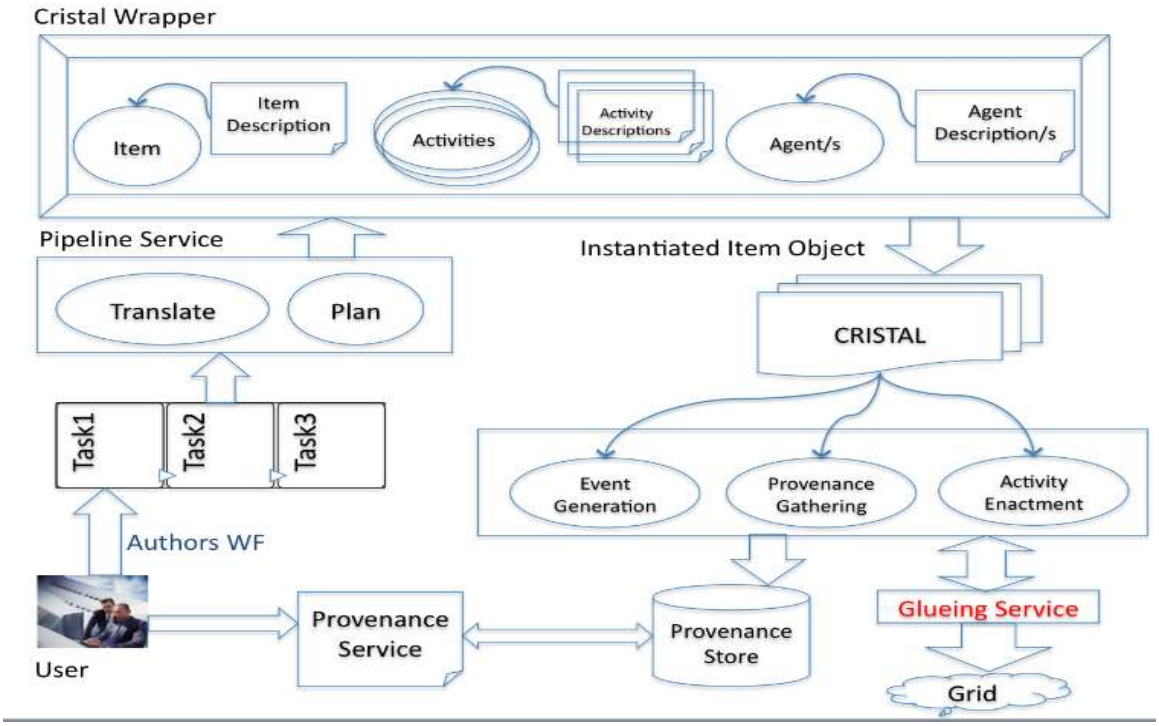


Figure 9: CRISTAL Architecture

The following sections describe the workflow generation, workflow coordination, provenance tracking, provenance storage and query processing aspects of the CRISTAL enabled provenance service. A sequence diagram of the whole process is shown in figure 10 that describes how various components in the CRISTAL wrapper will interact and coordinate with each other.

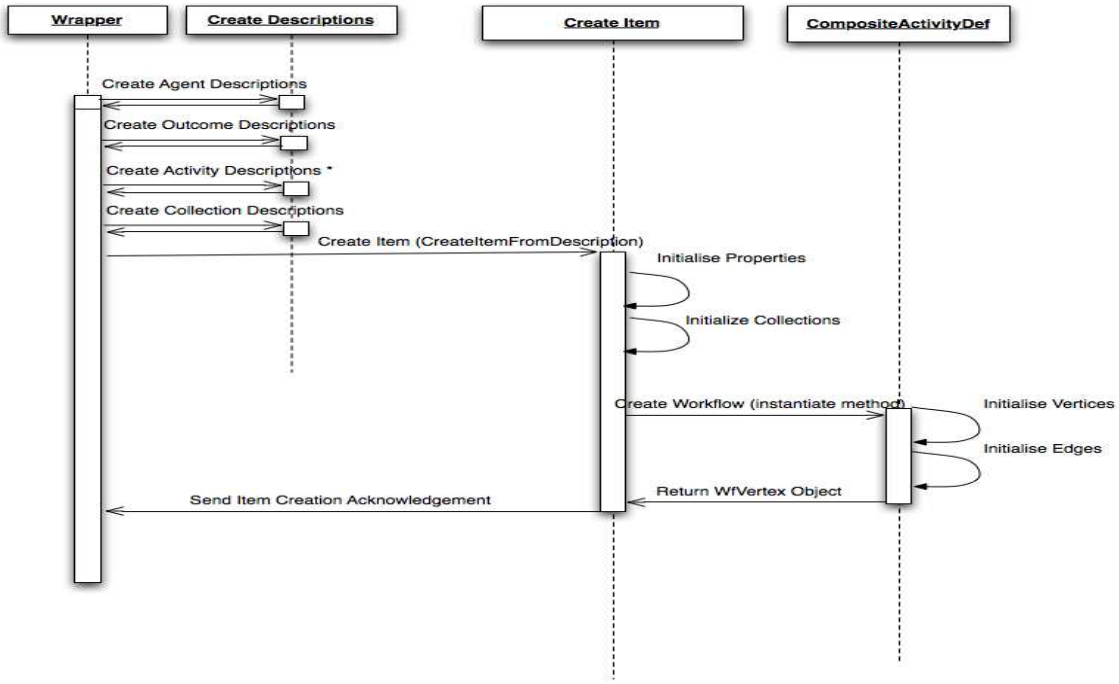


Figure 10: Sequence Diagram for CRISTAL-Wrapper Workflow Initialization

1. The CRISTAL Wrapper will first create so-called **Agent, outcome, activity** and **collection** descriptions of a workflow if they do not exist already. Agents in the CRISTAL model execute activities. In the neuGRID application and in a Grid environment in general, a compute element is directed by an agent to execute the workflow activities. In neuGRID a single *Agent* will be created for each of the users. It will take tasks and dispatch them to the Grid on behalf of the users.

To create agents in the Glueing Service, a domain specific implementation of the class *UserCodeProcess* must be provided. The method *runUCLogic* must be overridden to incorporate logic that would enable the submission of a job to the Glueing Service. Another change that will be required is to override the *assessStartConditions* method. In neuGRID complex and highly parallel workflows will be executed, hence each task may have multiple start conditions that must be fulfilled before execution can commence. The *assessStartConditions* method forces the *Agent* to do some pre-processing before starting an *Activity*. This processing could be authentication, authorisation or another form of dependency check. Some logic could be put in the *assessStartConditions* that will check if all dependent jobs of this workflow have been executed, or that a new thread should be invoked for another parallel branch.

2. After the descriptions have been generated, the *Item* can be instantiated through the *CreateItemFromDescription* class, which initialises an item to a specific *Domain Path* and creates a *System Key* that identifies the *Item*. The properties and the collections of the Item are defined and a workflow is specified which comprises the activities that have been previously described. In CRISTAL a workflow is modelled as a composite activity. A workflow may consist of other composite activities. The composite activities that are generated can be reused to create new workflows. The *CompositeActivityDef* class handles the definition of a workflow.

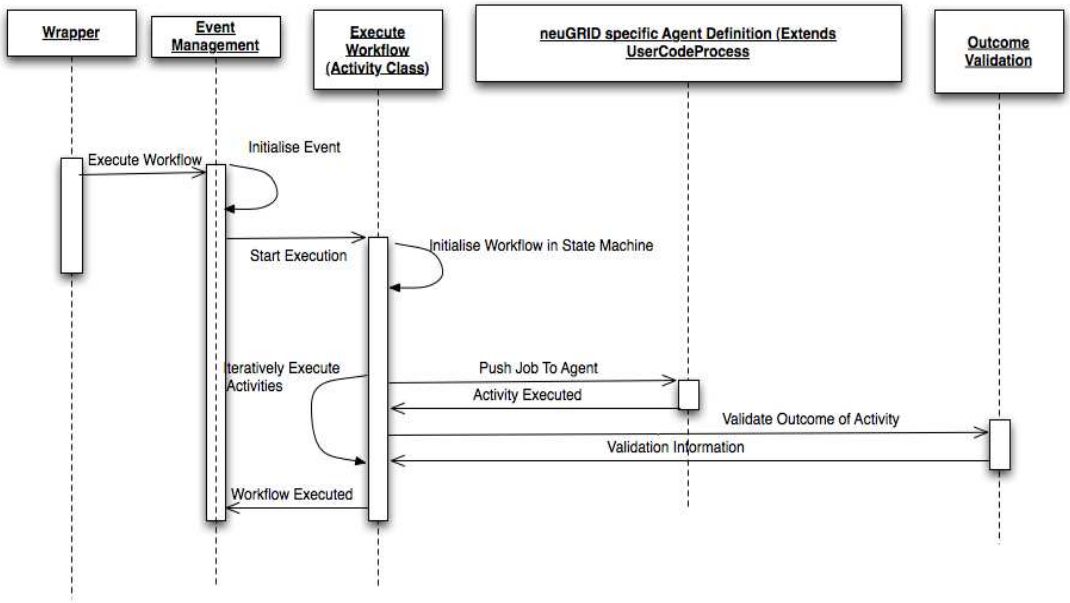


Figure 11: Workflow Execution

An item in CRISTAL is stored as a binary CORBA object. Communication with the CORBA object is handled via an *ItemProxy* object. The *ItemProxy* object is initialised with a CORBA IOR reference which identifies the Item in the CORBA server. Every time a workflow is executed an event is generated which stores the outcome of the workflow. An item will have several events in case a workflow is executed a number of times. Workflow execution in CRISTAL is handled as shown in the sequence diagram in Figure 11.

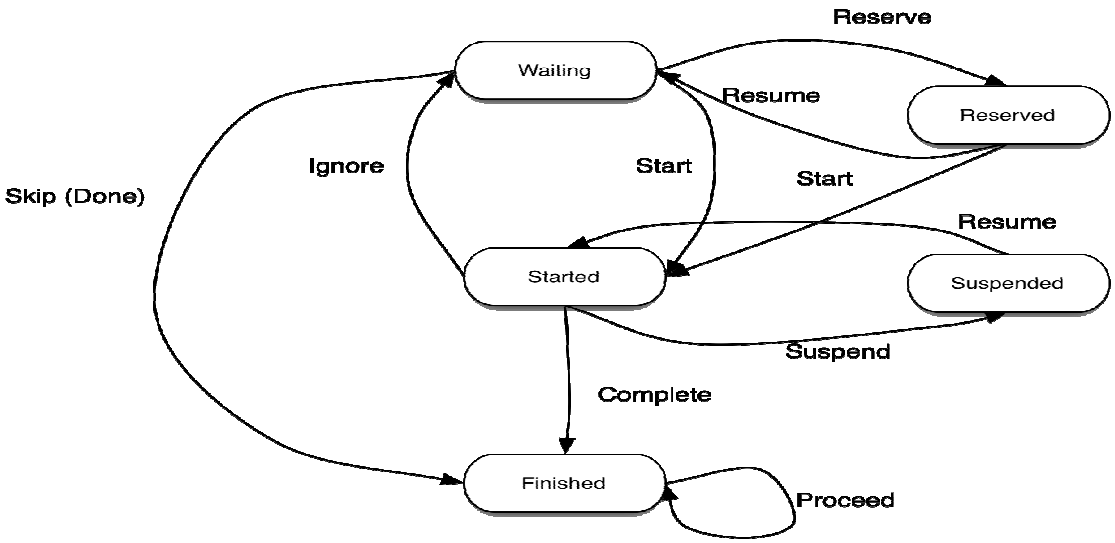


Figure 12: State Machine transitions for Jobs in CRISTAL

To execute a workflow a new **Event** is generated. The event generation initiates the execution of the workflow by initialising an ***Activity*** Object. The **Activity** object at first initialises a state machine of the workflow. The state machine tracks progress of an Activity's state while it is being executed. Activities can be described with a certain number of states such as "Suspended", "Interrupted" etc. The full set of state transitions supported in CRISTAL is shown in figure 12. The state machine iteratively executes activities of the workflow and events are stored at each activity state transition. An Activity is executed when a job, which represents an activity, is pushed

to its Agent. The Agent executes the activity according to its ***runUCLogic*** function. The outcome of the activity is validated against an outcome schema that is defined at the time of activity creation. Once the validation is successful the state machine iterates to the next activity to execute the next job. If the validation fails the execution of the workflow is halted.

2.4 Workflow Provenance

As previously discussed, CRISTAL generates events during the execution of a workflow. Figure 13 shows the workflow execution mechanism from a provenance viewpoint. The flow of recording an Activity state in CRISTAL is as follows:

1. When the Activity class initializes a state machine that executes the workflow, at each state transition for each activity, an Event is generated and stored. Objects are stored in a domain specific CRISTAL storage schema. The storage and retrieval of Items is determined and modified by configuring the ClusterStorage class. Events consist of the following attributes:
    a. *Name of the agent, who executed the current event*
    b. *Role of the agent*
    c. *Transition information*
    d. *Name of the step for the element in the Item (workflow)*
    e. *Path of the element in LDAP server*
    f. *The type of the step i.e. start node of the workflow or intermediate*
    g. *State information of the element*
    h. *Creation date*

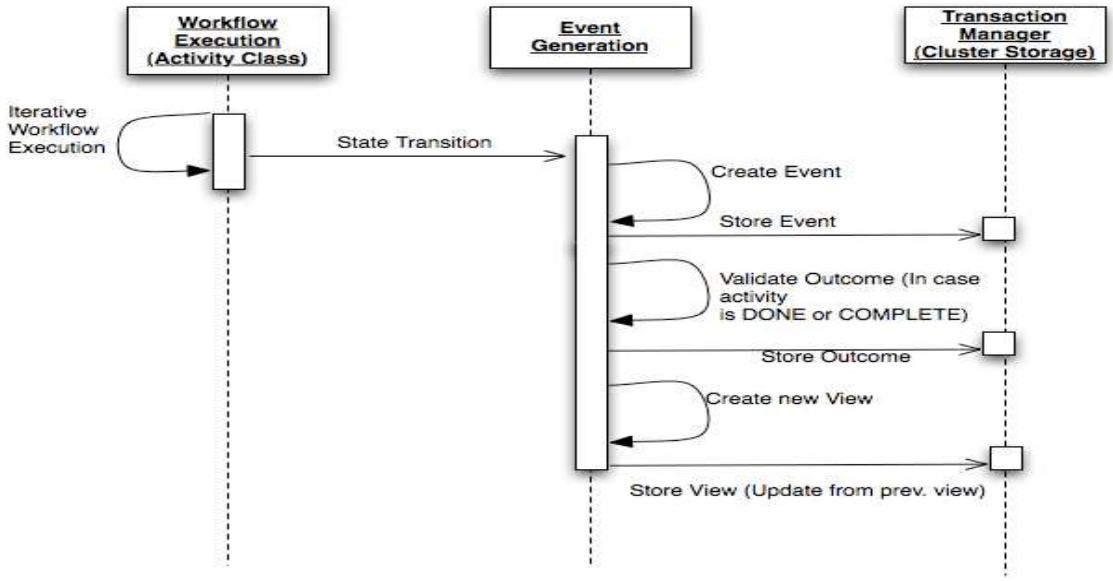


Figure 13: Provenance Recording in CRISTAL during execution

2. If a state transition occurs and the transition was caused by the completion of an activity, the outcome of the activity is first validated against an outcome schema and stored upon successful validation. In case the validation fails, the workflow execution is stopped. By querying the Events generated during the workflow execution users can trace what went wrong during an activity execution.

3. Another item that is stored at each transition is the view. A view represents a snapshot of the current version of the latest Event. In case a user wants to query for the last event created by the execution of a workflow, the user retrieves the “last” view.
4. After this step, a post recording check is performed to see if the current view already exists. In case it exists, a new version of the view is created; otherwise the versioning information of existing view is updated.

The recording of all events in an *Item* forms the history of the workflow along with description of the Item, properties, collections and outcome schema. The history maintenance in CRISTAL is shown in figure 14. *Item* objects in CRISTAL are stored in the form of binary CORBA objects in CORBA Server. By providing a CORBA IOR reference, a user can retrieve an object in CRISTAL’s client view. All the description, properties, collection and outcome objects are stored in an LDAP schema. CRISTAL uses CASTOR APIs for translating the object-oriented representations of these objects to XML files and these files are eventually stored in OpenLDAP. In the client view a user can retrieve these objects by providing a logical path of the LDAP Server. As the objects are flattened into simple XML files, at the time of retrieval, it reforms the object from XML description to present the information to users.

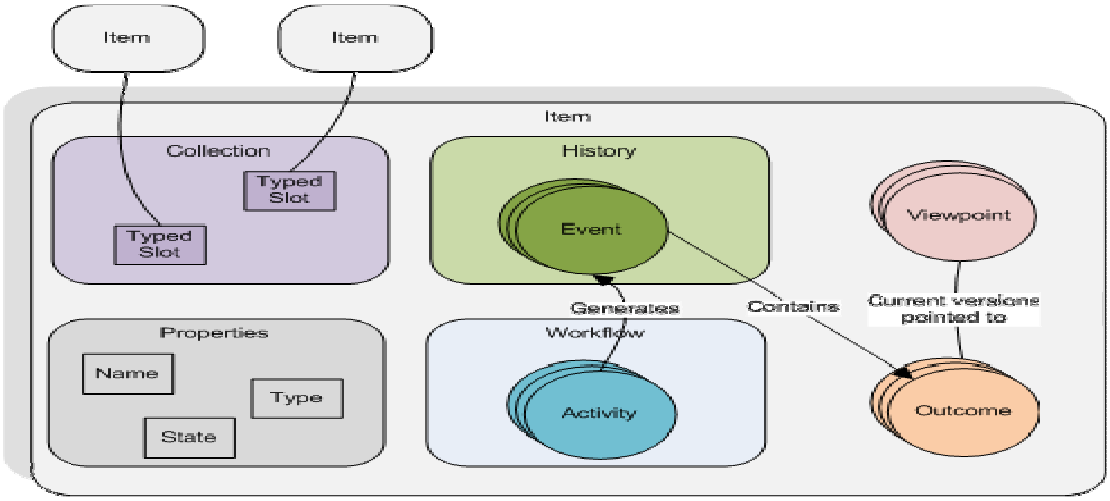


Figure 14: Item Structure

CRISTAL also has pluggable data storage support. This can be extended by replacing its default storage mechanism that stores XML files that can be queried via OpenLDAP. The key interface for this purpose is *ClusterStorage*. This interface uses a configuration file to connect to different relational databases. The default distribution uses *defaultConf.properties* file for creating an XML database in OpenLDAP directory structure. *ClusterStorage* provides six main functions and these functions must be overridden for using a specific database. These functions are listed below:

- *open() - initialization*
- *getClusterContents() – directory contents*
- *get()*
- *put()*
- *delete()*
- *query() – to expose underlying query engine*

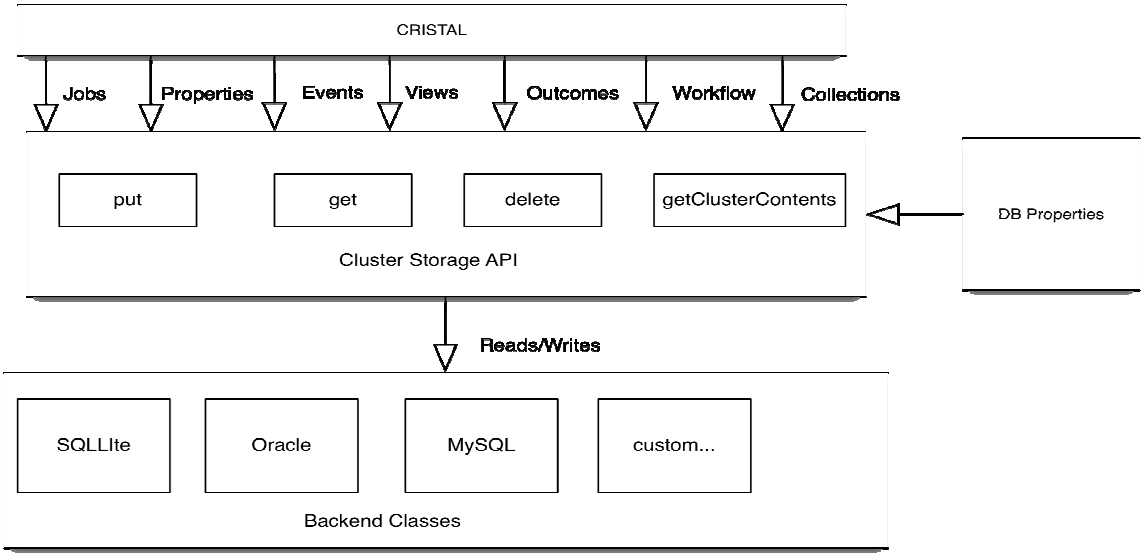


Figure 15: ClusterStorage API for provenance storage and retrieval

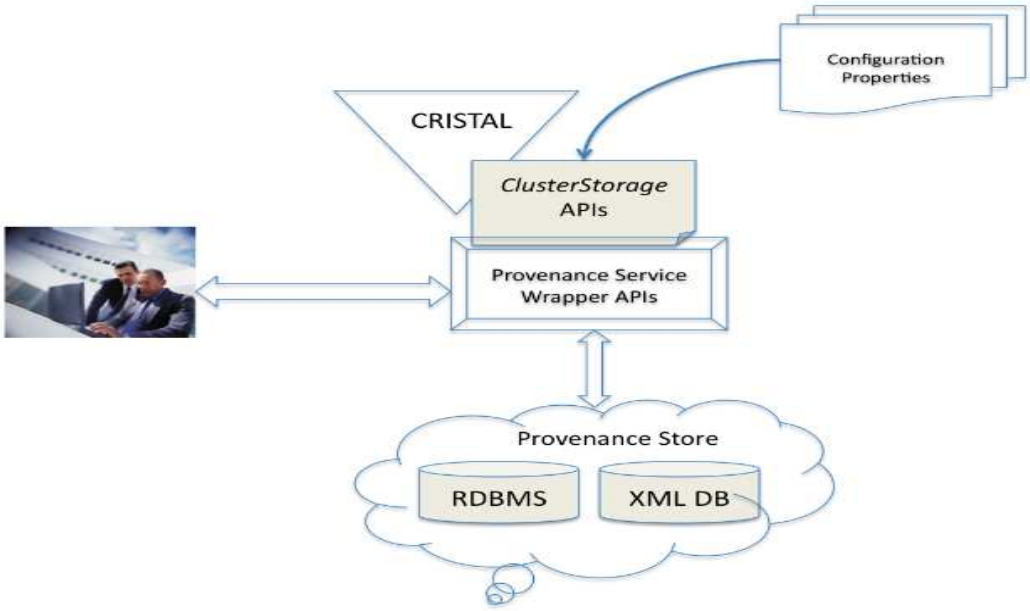


Figure 16: Provenance Service Structure

In order to implement *ClusterStorage* for a domain, one must override the *ClusterStorage* put, get, delete and *getClusterContents* methods, as shown in Figure 15. CRISTAL uses *ClusterStorage* to store properties, events, views, outcomes, workflows and collections for each Item. The Item itself contains "*Paths*" to these elements, which CRISTAL accesses through the *ClusterStorage* API. In the context of neuGRID project, *ClusterStorage* interface will be wrapped by Provenance Service APIs, which will provide extended functionality for recording and querying provenance information. The figure 16 shows this process in detail.

2.5 Future Directions

1. Integration with the pipeline service

   The CRISTAL structures should understand the structure of neuroimaging pipelines and scientific workflows in general. It should allow users to capture, browse, reconstruct and validate the pipeline related provenance information. The current functionality was not implemented for scientific workflow provenance and therefore this feature needs to be extended to allow an improved workflow support.

2. Authentication and authorization of the provenance data

   In neuGRID each user will have separate authentication and authorisation credentials. Therefore the issues such as granularity of authorisation and synchronisation of the CRISTAL data security with the security deployed in the rest of neuGRID need to be further explored.

3. Integration with the Glueing service

   The Glueing service provides a middleware independent mechanism to access resources and submit jobs. CRISTAL should expose suitable interfaces that can allow applications to make use of Grid/Cloud resources through the Glueing service. This approach will not tie CRISTAL down to a particular application or middleware platform.

4. Provenance Schema and database

   The current schema and database access mechanism needs to be refined to provide a fine grained querying and storage mechanism. CORBA related dependencies need to be removed. The provenance information may be stored on remote databases, which will have to be accessed through SOAP or similar protocols and such support is necessary in CRISTAL.

5. Provenance reconstruction

   The current provenance reconstruction mechanism in CRISTAL is not sufficient to enable the scientists to reconstruct their workflows. The reconstruction process should help in observing the pipeline creation process, re-executing a pipeline or part of it and modifying a pipeline and storing it with a different version.

3. The Glueing Service

3.1 Introduction

The Glueing Service is a constituent service of the generalised middleware services layer in neuGRID, which aims to provide:

1. A standard way of accessing Grid services without tying services and applications to a particular Grid middleware.
2. A mechanism to access any deployed Grid middleware through an easy-to-use service.
3. A solution that extends and enhances the reusability of already developed services across domains and applications.
4. A service-based approach to shield users and applications from writing complex Grid-specific functionality. The user requires a minimum set of Grid-specific APIs and the rest of the functionalities are managed by the service.
5. A simplified approach for enabling clients to interface/connect their applications with Grid infrastructures, without installing and maintaining too many Grid specific libraries.

3.2 Architecture

The Glueing Service exposes SAGA API functions as web service methods with a one-to-one correspondence. The client applications can transparently access the Glueing Service by using a SAGA SOAP adaptor, which we have named UWESOAPAdaptor. It is an implementation of the Adaptor API provided by SAGA. The client can include the UWESOAPAdaptor and write applications using the standard SAGA API classes. The SAGA API calls, generated on the client side, are passed to the Glueing Service by the UWESOAPAdaptor, which is responsible for communicating with the Glueing Service. The Glueing Service itself is implemented in Java and runs within a Tomcat server instance.

The UWESOAPAdaptor is a component between the client applications and the Glueing Service. The client applications define, create and submit jobs according to the standard specification of SAGA. These instructions are then translated into SOAP requests by the UWESOAPAdaptor. The SOAP requests are used for communication with the Glueing Service. The Glueing Service actually executes the client instructions using SAGA APIs and middleware adaptors.

The UWESOAPAdaptor requires a Service Endpoint URL to communicate with the Glueing Service. The Endpoint URL is used to access the service WSDL, which is then used for service invocation. The WSDL describes the definition of all the exposed methods. The UWESOAPAdaptor calls the published methods using SOAP requests and the Glueing Service sends back SOAP responses to the UWESOAPAdaptor. The UWESOAPAdaptor then translates the SOAP response and returns the execution results to the client in the form of Java or SAGA specific objects.

The following diagram (Figure 17) shows a scenario where the UWESOAPAdaptor interacts with the Glueing Service. The UWESOAPAdaptor passes the middleware

and job information to the Glueing Service using SOAP objects and the SOAP response is sent back to the UWESOAPAdaptor from the Glueing Service. The exposed functions of the Glueing Service are discussed in detail in section 3.

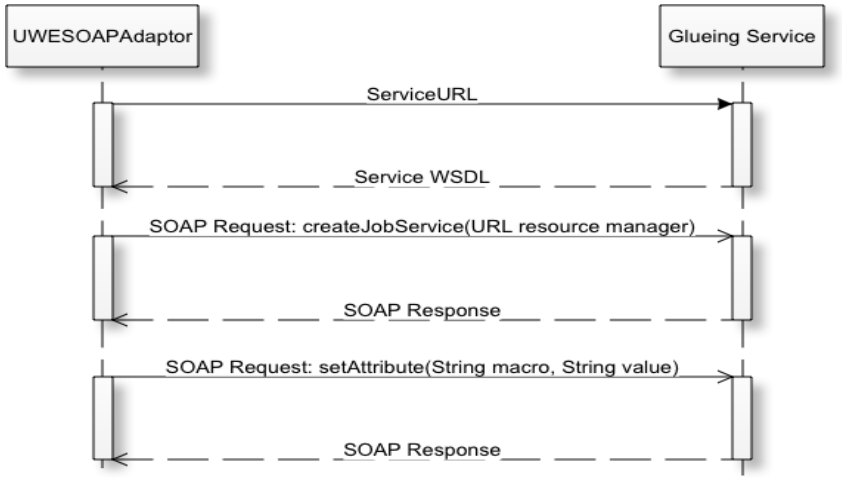
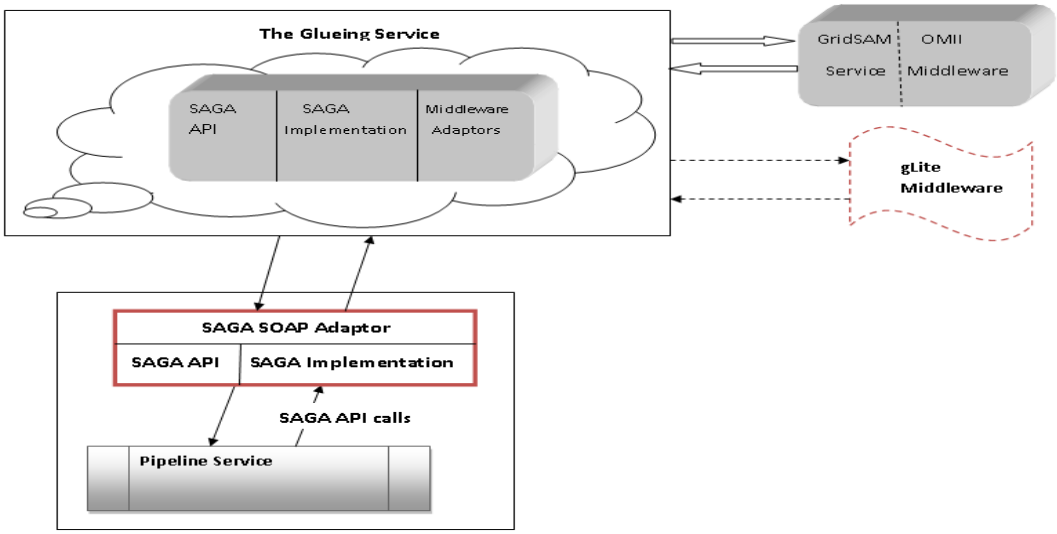


Figure 17: A sample use case

The architecture diagram (Figure 18) shows how an application contacts the Glueing Service. The Pipeline Service, shown in the figure, is one of the potential applications of the Glueing Service. This service uses the SAGA APIs and the UWESOAPAdaptor to communicate with the Glueing Service. The UWESOAPAdaptor accesses different methods of the Glueing Service by getting its WSDL. The methods, published in the WSDL, execute the actual instructions that are generated on the client side. Thus, the Pipeline Service initiates a grid activity that is then forwarded to the Glueing Service by the UWESOAPAdaptor.



.

Figure 18: Glueing Service Architecture

The Glueing Service, as shown in the diagram, can communicate with different Grid middleware such as OMII/GridSAM or gLite. This allows client applications, such as the Pipeline Service, to use Grid resources provided through different middleware. The Glueing Service is built on Java-SAGA, which is a java implementation of SAGA specification, and middleware specific adapters to communicate with a particular middleware. We are also using an adaptor for javaGAT that has been written to access the gLite middleware. The adaptor is discussed in section 3.4

3.2.1 Service Interface

The webservice interface of the Glueing Service is a set of SAGA-like methods that are exposed to clients. The signatures of some of the exposed methods are given below:

public String write(DataHandler dh, String fileName)

public String[] listFiles(String dir) throws Exception

public DataHandler read(String fileName)

public String runSAGA_OE(String url, String app, String args, String err, String output)

3.3 Functionality

The current implementation of the UWESOAPAdaptor encapsulates the following functionality:

- Job Submission
- File Management (Reading, Writing, Directory listing, Getting file size)
- Job Monitoring
- User Authentication

These are explained in detail below.

3.3.1 Job Submission

The client applications, using the UWESOAPAdaptor, pass job information to the Glueing Service. The Glueing Service (GS) loads the appropriate middleware adaptor, based on the site-specific server configuration. For example, if the site administrator has configured the GS to use gLite, the GS loads the gLite adaptor by creating a “Preferences” context for the JavaGAT adaptor. We are using the gLite adaptor that comes with JavaGAT, as explained in section 3.7. A context is a specific piece of information that is shared within a particular session. An application may associate different contexts with a particular session in order to make them available throughout the lifetime of the session and to all objects that are part of that session.

The GS tells JavaGAT to use gLite by setting the property “ResourceBroker.adaptor.name” to “gLite” for the “Preferences” context. Once this is done the Glueing Service creates a job, based on the job information sent by the client application. The Glueing Service uses SAGA job management APIs for creating a job, submitting it to the available Grid resources and retrieving its status. The SAGA job management API covers four classes; these are *JobService*, *JobDescription*, *Job* and *JobSelf*. These are explained below:

3.3.1.1 JobService (Selecting a Resource Manager)

The *JobService* API is used to select a resource manager. An instance of *JobService* represents a resource manager backend. A resource manager is an endpoint where the job is submitted by the client application. This resource manager can also be an execution service if it executes the job.
Input parameter: To create an instance of *JobService* class an endpoint URL of the resource manager is required as an input parameter to *createJobService* method.
Example: The Glueing Service uses an endpoint URL for submitting the job to the resource manager. For example if the resource manager is GridSAM then an instance of *JobService* is created as: *<JobService> js = JobFactory.createJobService (new URL ("https://HOST:PORT/Gridsam/services/gridsam").*

3.3.1.2 JobDescription

The *JobDescription* API defines the job using a well defined set of attributes such as the application executable and associate arguments. The *JobDefinition* attributes behave like tags in JSDL/JDL and thus these attributes mimic JSDL for the middleware and are passed to it internally by SAGA.
Input parameters: The *JobDescription* API needs two essential parameters in order to define the job i.e. the application executable path and the exact application parameters.
Example: If the application execution is "/bin/echo" which takes a string as input parameter i.e. "hello" then the Glueing Service, using JobDescription, defines a job as:
<JobDescription> jd.setAttribute (JobDescription.EXECUTABLE, "/bin/echo") and <JobDescription> jd.setVectorAttribute (JobDescription.ARGUMENTS, new String[] {"hello"}).

3.3.1.3 Job (Job Creation)

The *Job* class represents an actual job that can be submitted to the underlying grid middleware. An instance of the *Job* class can be created using *JobService.createJob*, which takes an instance of the *JobDescription* class as an input parameter. Instances of both *JobService* and *JobDescription* classes are pre-requisites for creating a *Job*.
Example: A job is created with:
<Job> j = <JobService> js.createJob (<JobDescription> jd)

The job thus created can be run with:
<Job>j.run()

3.3.2 File Management

The adaptor also supports reading and writing files to the middleware backend of the Glueing Service. To transfer files to the Glueing Service the adaptor uses the Java Activation Framework. This functionality is described in more detail in section 3.2.5. Once the file has been written to a temporary location, the Glueing Service loads the specific middleware adaptor based on the site-specific configuration, as in the case of job creation. The file is then written to the middleware backend using the loaded adaptor. It should be noted that because the Glueing Service is in fact a stateless web service, writing to or reading from files in chunks cannot be supported. It would be prohibitively expensive to provide such functionality.

3.3.2.1 Writing a file

To write a file to the Glueing Service, the client first creates a *File* object using:
<File> f = FileFactory.createFile(path-to-file, Flags.READ.getValue())

The READ flag is important here because it tells the adaptor that the client is getting ready to read a file and write it to the Glueing Service. Then the file is actually written with:
<URL> u = URLFactory.createURL(path-to-file-on-GS)

f.copy(u)

f.close()

3.3.2.2 Reading a file

To read a file from the GS, the client creates a *File* object:
<File> f = FileFactory.createFile(path-to-file-on-GS, Flags.CREATE.getValue())
The CREATE flag tells the adaptor to create a local empty file in anticipation of the one that will be read from the GS. The file is then read with:
<URL> u = URLFactory.createURL(local-path-to-file)
f.copy(u)
f.close()

3.3.2.3 Directory listing

The following code lists the contents of a directory:
Directory dir = FileFactory.createDirectory(session, serverDir);

System.out.println(dir.list());

dir.close();

3.3.2.4 Getting file size

To get the size of a file, a client must do the following:
File remoteFile = FileFactory.createFile(session, remoteURL, Flags.CREATE.getValue())

System.out.println("Remote file size: " + remoteFile.getSize() + " KB")

3.3.2.5 Handling SOAP attachments

Files are transferred to the GS as SOAP attachments using Axis2J. Axis2J uses the Java Activation Framework to handle the transfer. To upload a file to the GS, the following interface is used:
private String uploadFile(DataHandler dh, String fileName)

The UWESOAPAdaptor uses the following code to transfer the file:
Write writer = new Write();

DataHandler dhSource = new DataHandler(new FileDataSource(nameUrl.getPath()));

writer.setDh(dhSource);

writer.setFileName(target.getPath());

```
try {
        gs.GlueingServiceStub stub = new
```

```
        gs.GlueingServiceStub( sessionImpl.listContexts()
        [0].getAttribute("ServiceHost") );
        WriteResponse response = stub.write(writer);
        logger.info( "Method copy returned with: " +
        response.get_return() );
}
catch (java.lang.Exception e)
        {
                e.printStackTrace();
                throw new NoSuccessException("Unable to write file to
                server.", e);
}
```

3.3.3 Job Monitoring

To monitor the execution of a job, the client must register a callback with the adaptor. The callback class must implement the *Callback* interface. To register the callback:
<Job> job.addCallback(metric, callback object)

The *Callback* interface specifies the following method signature:
public boolean cb(Monitorable m, Metric metric, Context ctxt)

3.3.4 User Authentication

In its current implementation, the adaptor supports the gLite user authentication. An application developer creates a *"Preferences"* context and passes it on to the UWESOAPAdaptor. The context contains various settings for the gLite adaptor that it requires to function. The some of the attributes are listed below:

- VirtualOrganisation
- vomsHostDN
- vomsServerURL
- vomsServerPort

To specify the gLite certificate, the user creates a *"Certificate"* context. This context contains the following attributes:

- Context.USERCERT
- Context.USERKEY
- Context.USERPASS

Context.USERCERT and Context.USERKEY contain the locations of the user certificate and private key respectively. The UWESOAPAdaptor uploads both these files to a temporary location on the GS, where they are stored until the gLite adaptor needs them. The locations of these files are then passed to the gLite adaptor for authentication. When the adaptor no longer needs the files, they are removed from the GS.

As previously said, all this mechanism will be removed as soon as the SSO integration will be done.

3.4 gLite Adaptor

Starting with version 1.0.1, a gLite adaptor has been introduced into the SAGA Java implementation. This adaptor supports job submission and monitoring. However, at the moment it only supports submission of single jobs. Since the SAGA API currently does not support workflows, the gLite adaptor has been extended in a non-standard way to support workflows.

We have provided extensions to the Glueing Service that will directly interact with the JavaGAT engine for the workflow functionality. In this case we have used SAGA for all the functionality SAGA provides, and we have extensions at the Glueing Service level for functionality which SAGA currently does not cater for. We do not want to break the SAGA standard, and therefore implement functionality "apart" from SAGA at the Glueing Service level in order to implement WP6 services philosophy. In future however, as workflow support is added to SAGA, we only need to change the Glueing Service to migrate functionality from the JavaGAT engine to SAGA implementation. This would not affect any other Service at all. When it will be officially available inside SAGA, we will simply deprecate this functionality.

3.5 LORIS Integration

We are in the process of testing and integrating the UWESOAPAdaptor to the Prodema LORIS application. In this regard, we are working closely with the LORIS developers and initial integration and tests have been successful. Since we do not have direct access to the LORIS source code, we have to rely on the LORIS developers to test the adaptor functionality and provide feedback. Once the required features have been fully tested, we plan to roll out the UWESOAPAdaptor in the neuGRID production environment.

3.6 Limitations and Issues

The Glueing Service exposes SAGA APIs; therefore it can only provide those functions that are supported by SAGA API. The requirements are not fully addressed in the current implementation of the Glueing Service because of the lack of support for those requirements in the current SAGA implementation. Some important limitations in the implementation are discussed below:

3.6.1 Workflow Support

As the Pipeline Service generates pipelines or workflows to be executed over the Grid, it needs an enactment engine that can break the workflow into its constituent parts/jobs. It also needs to resolve job dependencies and then execute the sequence of jobs efficiently. The current release of SAGA can only submit one job at a time, through its JavaGAT adaptors, to a submission system such as GridSAM. Thus it does not have support for workload management and scheduling a series/sequence of jobs according to the requirements of workflow. This lack of workflow enactment in SAGA limits the scope of the Glueing Service. Thus, the Pipeline Service that deals with workflows cannot be fully supported by the Glueing Service at the moment. We have developed a temporary workaround for this, as described in section 3.4. This does not support reporting of the workflow status since the gLite adaptor was written

for single jobs, although gLite reports the status of the entire workflow as a single job. We are currently investigating various possible solutions that will allow the gLite adaptor to report the status of the entire workflow, as well as retrieve the output sandboxes for the jobs that constitute the workflow once they have been executed.

3.6.2 Single Sign-on

We need to implement a single-sign functionality to facilitate users. Currently to invoke enactment of a workflow and access other resources, the user's certificate, key and associated passphrase are required to initiate a proxy at the Glueing Service end. To use this model the user has to provide all of these details every time a requested is made. To cater for this limitation the glueing, pipeline and other services need to be integrated with a Single Sign On service (SSO). This task will be achieved in year 3.

3.7 Conclusions

The Glueing Service addresses major requirements of the neuGRID project and will provide a generalised framework for accessing resources over the Grid. The heterogeneity of distributed resources and details of grid middleware architectures will be transparent from users. The Glueing Service also hides complexities of interfacing with different grid middleware, which will allow accessing grid resources through a set of high-level functions. The service exposes SAGA APIs and can communicate with different middleware through their middleware adaptors. Details of middleware interactions are kept hidden from users enabling them to seamlessly use grid functionalities. This shields the low-level middleware difficulties from the user and will encourage them to use them with little or no knowledge. The design of the Glueing Service is based on SOA principles, which will help different services in neuGRID to use service functionalities through standardized interfaces. This will also allow other client applications to use service features by inspecting its WSDL, available online at the service endpoint URL. The SOA-based architecture of the Glueing Service is in line with the project requirements and will provide a gateway for all WP6 services to access grid resources.

4. The Querying Service

4.1 Introduction

The user requirements analysis clearly identified that heterogeneous sources of complex data are common in clinical research environments. The Querying Service is therefore an important service within the generalised middleware services layer. It will provide methods to enable the efficient querying of heterogeneous data in neuGRID. The primary aim of the service currently is limited to allowing users to query the data successfully. In the future we will explore the intelligence assistance to the user during query formulation. The data as well as being heterogeneous in nature could also be in many different formats. The Querying Service, as stated, is designed to accommodate heterogeneous data. This includes data formats that range from images, flat files, relational databases to XML. This service (as depicted in figure 19) will provide a choice of ways in which the user can query the data held in neuGRID, including:

- Create a querying service which can query disparate data resources. These data sources, in the context of the neuGRID project, are the provenance, LORIS and other repositories in the Grid where source as well as the analysis data may be stored.
- Craft a solution which is platform independent and service oriented.
- Where possible create a synergy between the querying of heterogeneous data resources and the associated metadata.

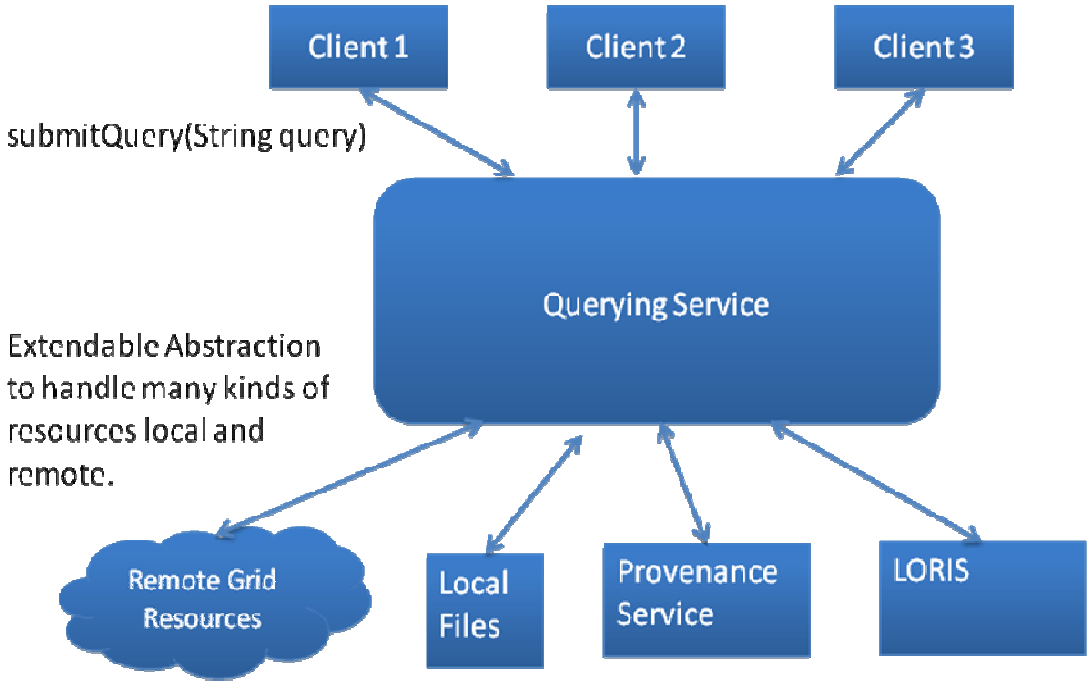


Figure 19: The Querying service will query a number of heterogeneous data sources

Year 1 deliverable D6.1 reported the work that was carried out at the early stage in the project to analyse the initial user requirements and identify a number of potential service models that could be implemented. Following this, a phase of prototyping and experimentation was put in place to gather as much information as possible prior to a final implementation strategy being created. It was felt that as the querying service depended heavily on the designs and implementations of other services, the development schedule for this service should be brought into alignment with the completion of the user requirements analysis. This allows the querying service to be tailored to better address the requirements of users and provides enough time for other system services to reach a level of maturity before any final decisions are taken.

4.2 Draft Service Model

The design shown in figure 20 was proposed in D6.1 as the candidate model for the Querying Service. Further experimentation and evaluation has confirmed this as a potentially good choice although it will be evaluated in the light of the final user requirements specification (D9.2,). Implementation details from the other services that will use the querying service will also be used to evaluate this model.

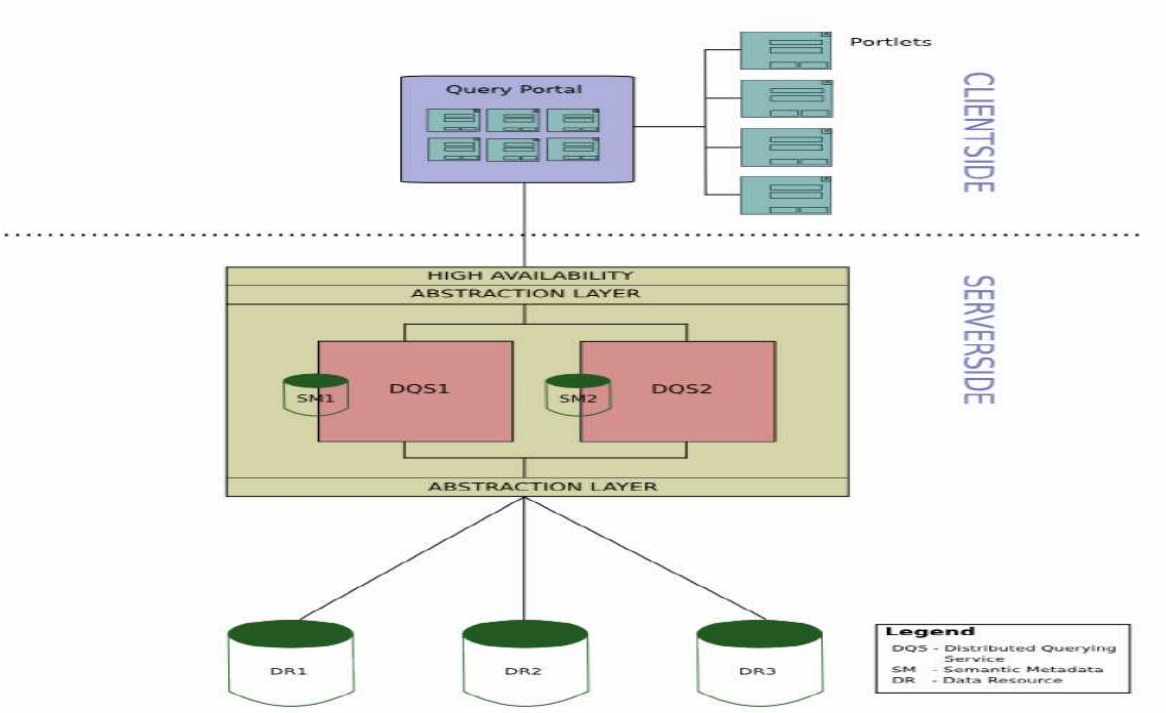


Figure 20: A Centralised Meta-data Approach

The design that was identified in D6.1 was initially selected because of its high availability. In this architecture, several instances are deployed simultaneously; this means that if one of the instances fails to respond initially, the client could automatically select a different Distributed Query Service (DQS) instance. The quality of service therefore would likely remain unaffected by such a failure. The distributed querying service could be instantiated multiple times and each time it is deployed, a local instance of the meta-data database is created. Queries may be more efficient with this architecture since each DQS holds a local instance of the meta-data.

Backup would be fairly trivial in the case of this model since the data is instantiated in several places at any one time and a master copy could be kept somewhere and

sequentially updated. Scalability is a requirement that this model well fulfils. If querying is extremely popular and the quality of service falls below expectations, the DQS could be deployed on another server, thereby making the data more available.

The model could be modified to provide only one instance of the DQS and a single instance of the meta-data. This would make maintenance easier but the high availability would be compromised and the model would not be as scalable to allow more traffic. Load balancing could be implemented as a layer above the querying service instances, with each service providing information to the load balancer based on their load and speed. The client could use this information to select the best choice (nearest querying service with the lowest load). Every time a query is submitted, it would first pass via the load balancer which would select the optimum querying service. The simplest way of implementing such a load balancer would likely contain a single point of failure. Clients could however, select a default querying service which is known to be available as a fallback in the case that the load balancer is down, thus eliminating this issue.

In summary, the draft service model offers the following advantages and poses the following issues.

Advantages:

· Well suited to an SOA design.
· Scalable.
· Straightforward to backup.
· High Availability

Disadvantages:

· Raises the issue of keeping the DQS instances up-to-date and consistent.
· Bandwidth is a valued resource with some institutions suffering from low levels and these could be put under strain if each query that enters the querying service goes to them.

4.3 Provenance Querying

Provenance querying is an important aspect of the querying service. CRISTAL already offers partial functionality to store and query provenance and we need to extend it so that the provenance repositories could be queried from the Querying Service. There are primarily two ways for querying the provenance information in CRISTAL that can be extended in the querying service.

One method is by using the querying service to *directly* access the database where *ClusterStorage* stores all the data. The domain specific implementation can be integrated with the neuGRID PKI based security infrastructure. To map workflows and Items and the associated Events to a specific user credential, a property might be introduced which holds the users Distinguished Name (DN). The details can be read in the provenance service section. When a user with a certificate containing the concerned DN accesses the Item, the user should be able to query and retrieve the results. This mechanism can be further extended to incorporate access control lists and implement fine grained authorization policies in accessing the provenance data. This interaction is shown in the following sequence in figure 21.

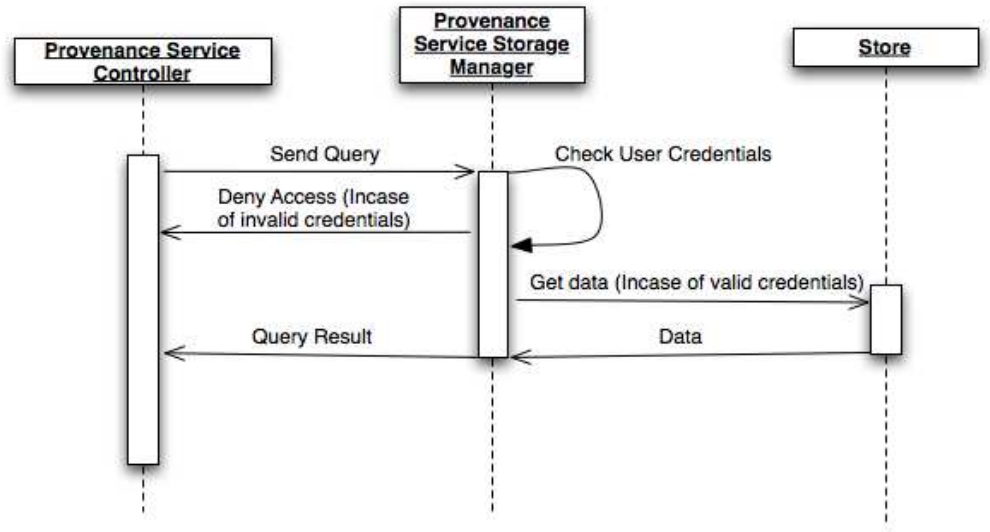


Figure 21: Query Processing through Provenance Service

The second method for querying the provenance information stored in a CRISTAL store needs a query to be sent to the particular *Item*. Users can communicate with Items via an *ItemProxy*. The querying sequence diagram for this particular case is shown in figure 22. In this architecture if a user desires to query for a certain set of events during the execution of a workflow, the user will first provide his credentials to the CRISTAL Wrapper through an appropriately defined API. The CRISTAL wrapper query method will use the CRISTAL Gateway API declared in *com.c2kernel.process.Gateway*. The Gateway API will allow the wrapper to retrieve an *Item* Object based on the CORBA IOR (identifier for the Item) and initialize an *ItemProxy* to communicate with the Item through the CORBA protocol. The Wrapper will then call the *queryData* method on the *ItemProxy* to retrieve query results.

In case the value of a certain property of the Item is to be retrieved the following query is specified:

• /*Property/[Name]*

If a workflow definition is to be retrieved the following query is specified:
• /*LifeCycle/workflow*

If an outcome is to be retrieved that was generated at a specific event, the following query is specified:

• */Outcome/[SchemaName]/[SchemaVersion]/[Eventid]*

To retrieve an event the following query is specified:

•*/AuditTrail/[Eventid]*

If Job information, along with state related information is to be retrieved, the following query is specified:

• */Job/[Jobid]*

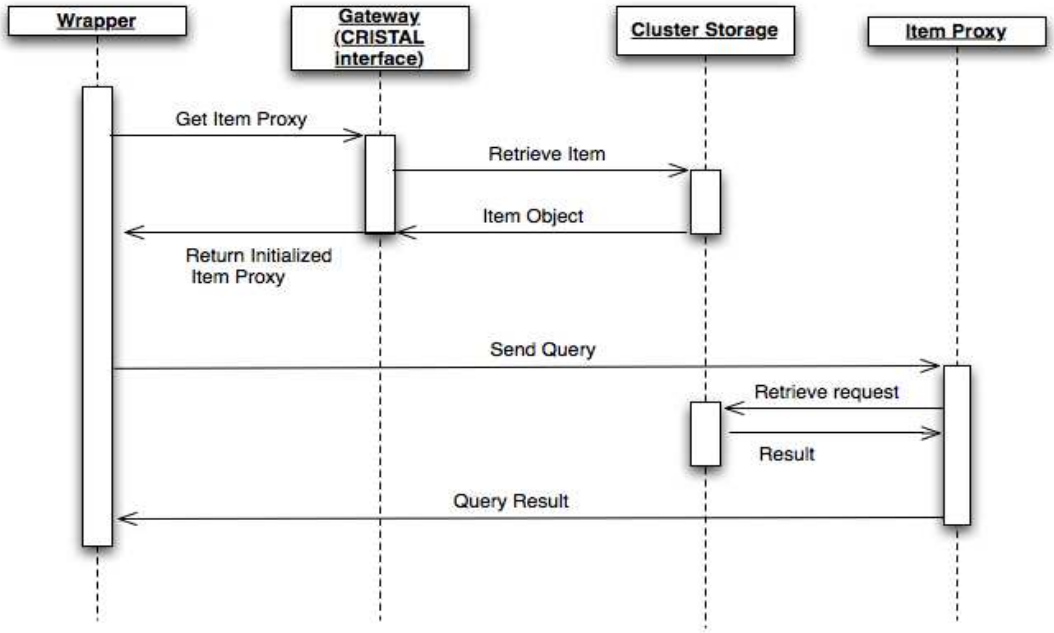


Figure 22: Query processing through ClusterStorage

4.4 Conclusions and Future Directions

A service was designed that is in line with the neuGRID design philosophy. The initial user requirements from WP9 were analysed and followed to produce a candidate service model. D6.1 reported this work and a phase of prototyping and experimentation followed, which allowed the gathering of information that will drive the final service implementation strategy. It became apparent that the Querying Service more than any other, depended on the designs and implementations of other services. The development schedule for this service was therefore realigned with the completion of the user requirements analysis. This will allow the querying service architecture to be informed by the requirements of users and provides enough time for other system services to reach a level of maturity before any final decisions are taken. In the coming months the query service architecture will be implemented to allow flexible access to neuGRID data. An API will also be developed forming a standard interface to the querying service. This process is timed so that the resulting querying service can be fully tested before it is integrated within the neuGRID infrastructure.

5. The Portal Service

5.1 Introduction

The Portal Service is the single point of entry for users to access the neuGRID services. It hides the complexity of the underlying low-level neuGRID architecture from the users and enables them to focus on using the services' functionality. It allows users to simply authenticate, access the services, browse the data, launch analysis and visualise their results.

The user requirements, which have been collected by the WP9, state the following functionalities have to be addressed by the portal service:

- Easy to use interface.
- Flexible interface that could be easily customized and reused.
- Single point of contact to access the underlying services that may be further composed with the different low level services.

The Portal Service is based on open standards to ensure re-usability and a good level of integration with other components. An important aspect is the federation of the existing web applications using a Single Sign On (SSO) facility and a shared neuGRID menu. This will provide the users a feature rich and a harmonised interface. The creation of a dedicated portal, which will aggregate services without an overly restrictive web interface, will enable users to add the missing building blocks to the portal. Thus neuGRID will offer a harmonised and federated set of specialised and dedicated interfaces to the users. It is anticipated that this will deliver a satisfactory user experience, thereby encouraging the wider adoption of the neuGRID platform.

5.2 Architecture Description

The architecture as shown in figure 23 is made up of three principal components: the neuGRID Single Sign-On (SSO) system, the neuGRID dashboard (menu – in orange) and the neuGRID JSR286/WSRP 2.0 compliant portal.

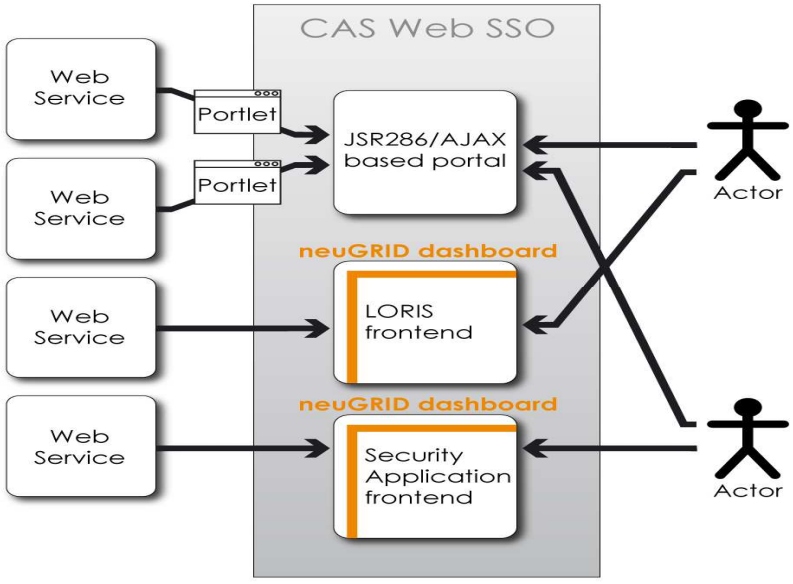


Figure 23: The Portal Service architecture.

A Single Sign On (SSO) system is a mechanism that allows a user to share its authentication between different applications. One account is used to access all the applications and one single login is required in order to access all the facilities. The neuGRID SSO is based on CAS, the Central Authentication Server, which is a widely used Open Source SSO implementation in Java. It has been adapted to the neuGRID architecture, and integrated with the MyProxy service. After a successful authentication (as shown in figure 24) CAS returns user's attributes as SAML assertions, so that protected applications are able to get all the required information.

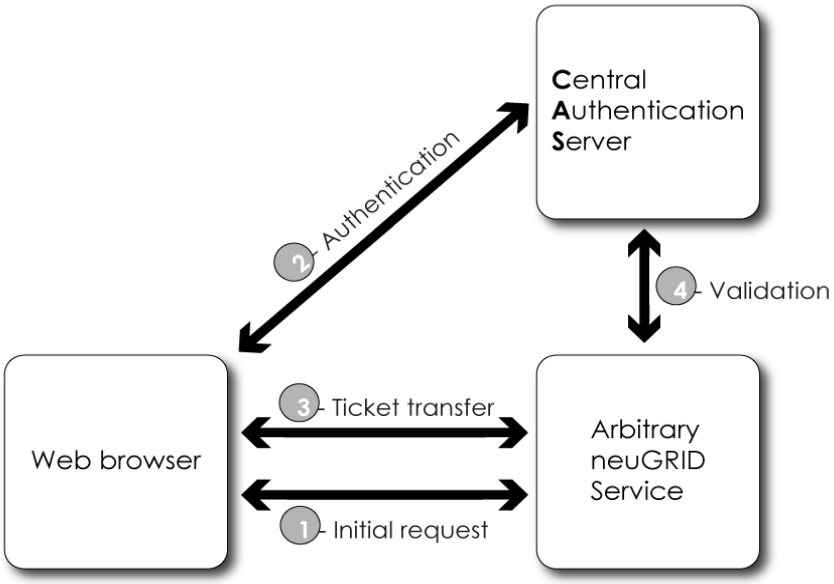


Figure 24: CAS Workflow

The neuGRID dashboard is a JavaScript menu that can be integrated into different services, federated by the SSO, providing them a unified neuGRID navigation menu. JSR 286, the Java Specification Request 286, is the Portlet specification created to enable interoperability between Portlets, Portals and Portlet Containers. These components allow the integration of mark up code from heterogeneous sources into a single portal. WSRP 2.0, Web Service for Remote Portlets version 2.0, is a recent OASIS Standard that introduces the possibility to integrate remote JSR 286 portlets (as shown in figure 25) into a portal alongside the local portlets.

The portal will integrate both local and remote portlets to aggregate the different features offered by the neuGRID web services. AJAX usage will limit client-server exchange and will allow refreshing only the required part/portlet of the page that underwent some change.

Portlets are small applications which have their presentation layer as a pluggable User Interface component. A portlet provides a specific piece of content (information or service). Each portlet that is present in a page produces markup to be included as a part of the portal page. The lifecycle of the portlets is managed by the Portlet Container which runs the portlets, providing them with a runtime environment and handling the persistent storage of the portlets' preferences. The Portlet Container receives request from the portal to execute requests on the portlets. A Portal, as shown

in figure 26, is responsible for the aggregation of the content received from portlets and provides facilities to authenticate users and personalize the content.

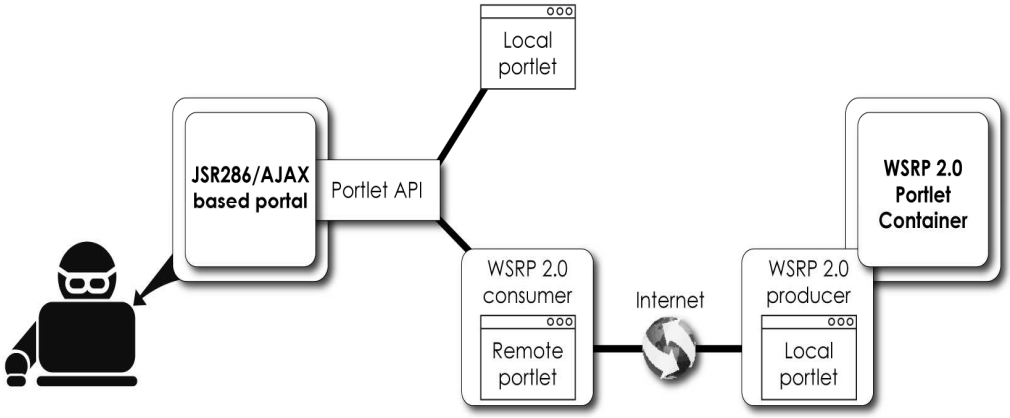


Figure 25: Overview of a JSR286/WSRP 2.0 architecture.

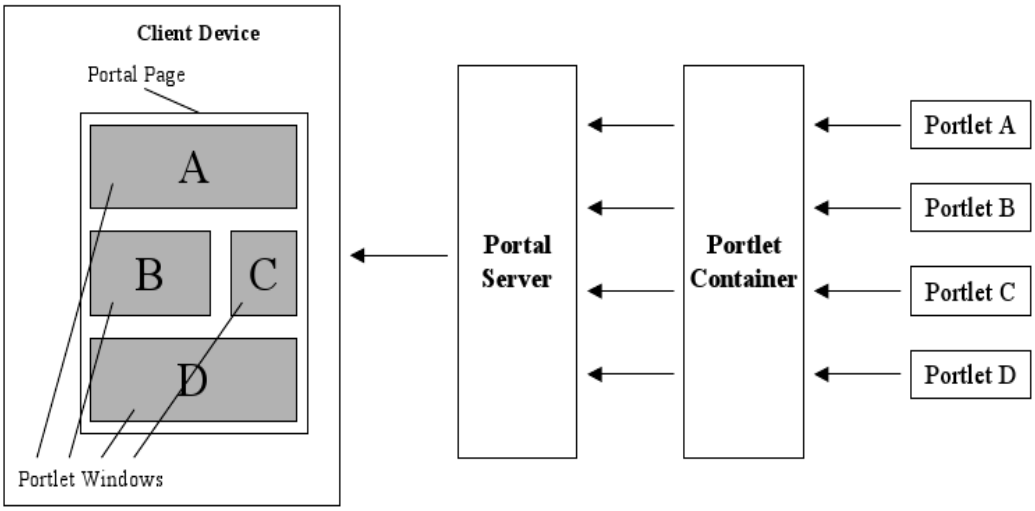


Figure 26: The Portal Page Creation - extracted from the Java Portlet Specification, version 2.0

Portlets follow the Model View Controller (MVC) pattern that separates the responsibility between a model, a view and a controller thus enforcing a clean application design. They have multiple window modes (VIEW, HELP, EDIT), and multiple window states (NORMAL, MAXIMIZED, MINIMIZED) accessible using the portlet's window controls. It is also possible to hide the controls, ensuring that the users see what the portal administrator wants them to see without being able to alter it. With JSR 286, as shown in figure 27, it is also possible to update only single portlet of the page.

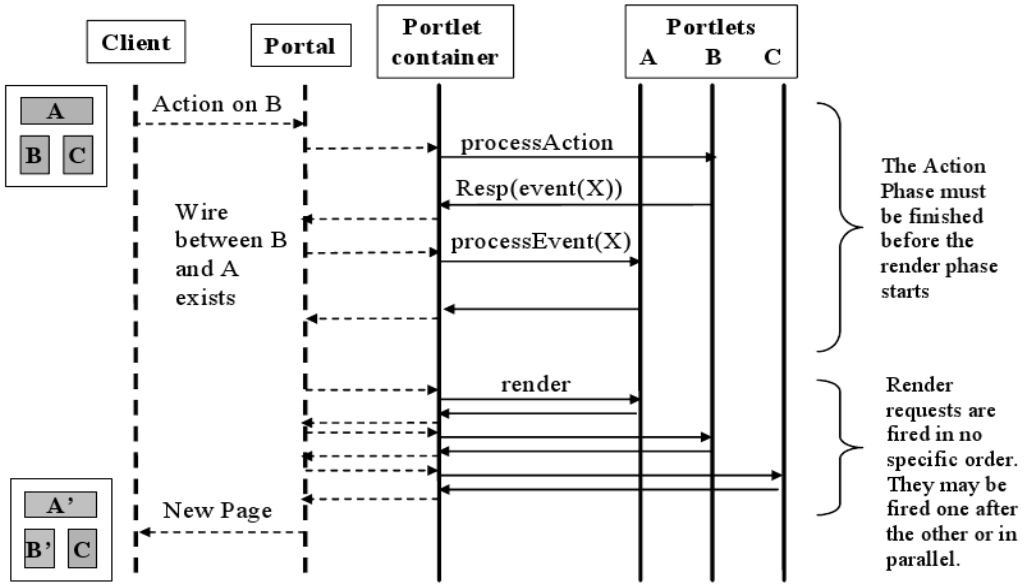


Figure 27: Request Handling Sequence - extracted from the Java Portlet Specification, version 2.0

The chosen JSR286/WSRP 2.0 compliant portal is Liferay Community Edition portal, which is an open source portal environment in Java. Integration with the CAS Single Sign On facility, which is used by neuGRID, can be easily achieved as Liferay is CAS aware. The OpenPortal Portlet Container 2.0 can also be integrated with Liferay. It's not yet the default Portlet Container, but once selected it allows the use of its WSRP 2.0 features, either as WSRP 2.0 Producer or WSRP 2.0 Consumer. The use of jQuery, a JavaScript based querying framework, allows creating nice and responsive interfaces as shown in figure 28.

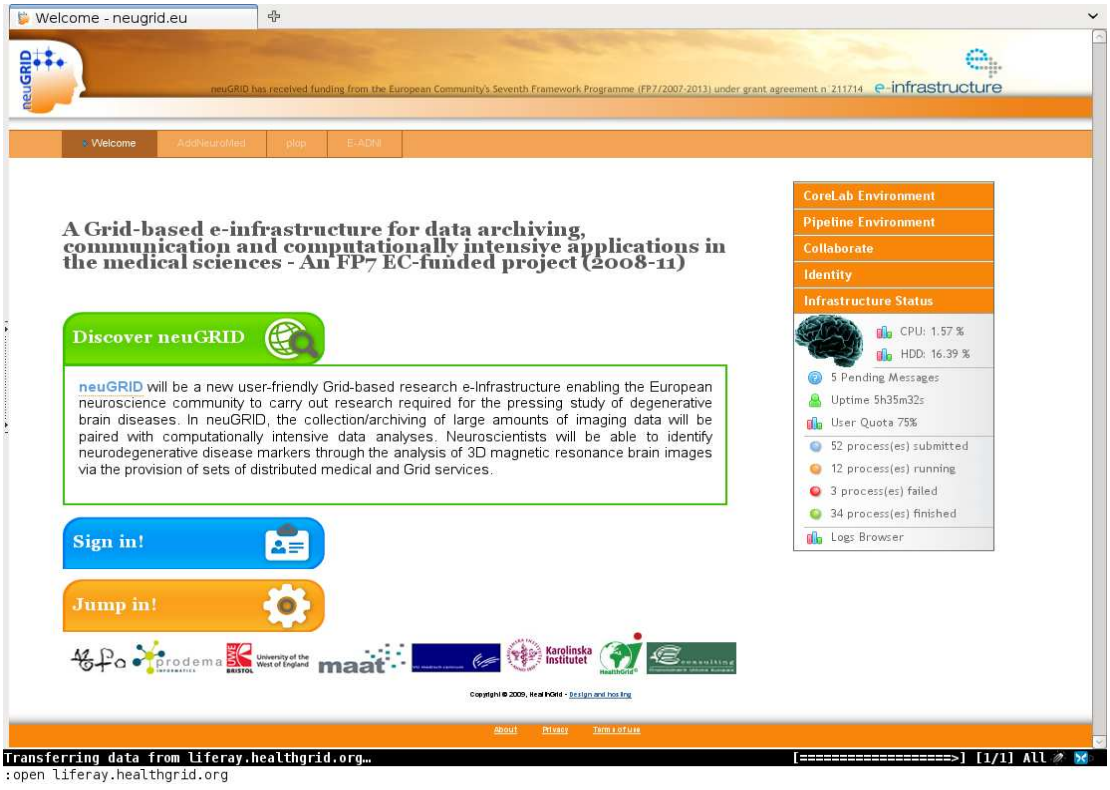


Figure 28: The Liferay-based neuGRID portal.

The architecture offers several benefits: it is standards-based, it allows decoupling user interface from web services and as a consequence it should be quite easily reusable. It even allows the reuse of existing portlets. It can be used to make the interface user-customisable and can provide users with their own set of private and public pages.

5.3 Functionality and Features

The Official CAS support is obsolete as Liferay is still using CAS Protocol 2 whereas neuGRID is using CAS Protocol version 3 and the SAML assertions that are extensively used in the neuGRID SSO architecture to provide access to the users' attributes to the different neuGRID services. It required the development of a custom authentication handler based on the existing CAS integration, using the features provided by Liferay. It was updated to handle the CAS 3 protocol. This step required to create the class neuGRIDAutoLogin which implements com.liferay.portal.security.auth.AutoLogin. This class is responsible for extracting the user's attributes from the SAML assertion that are returned by CAS once the user has been successfully authenticated. A Liferay user is created using these attributes if it does not already exist into the local Liferay database. The Distinguished Name (DN) of the user is normalized and used as the screen name.

A Liferay hook was also developed in order to be able to customise the terms of use page. A new portal user has to accept these terms before being able to use the neuGRID portal. The former neuGRID portal developed using Ruby On Rails has been completely migrated to Liferay. The design has been migrated as a Liferay theme. The neuGRID menu has been ported into a portlet including the Grid load glowing brain that is updated using Ajax and the resource serving facility of the JSR 286. The portal is now ready to receive the neuGRID portlets.

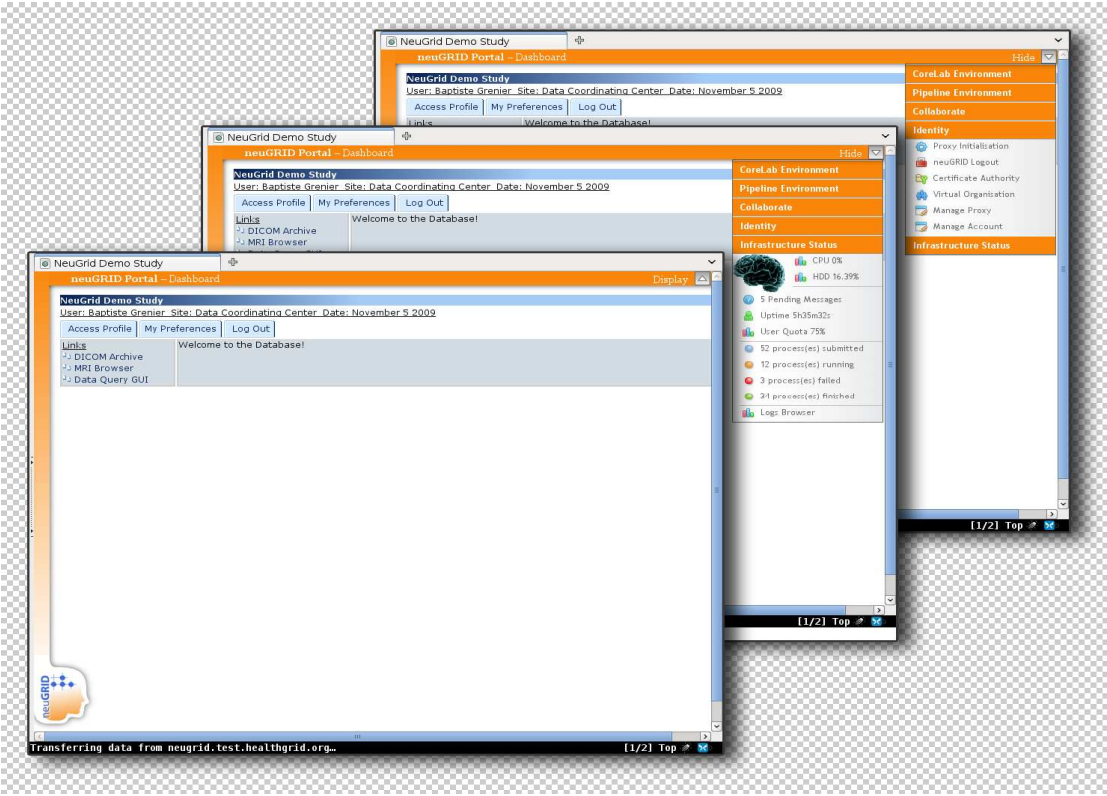


Figure 29: The dashboard integrated with the classical LORIS frontend

The neuGRID's dashboard (see figure 29), which allows to seamlessly integrate the neuGRID menu into any web application, is also integrated with CAS. With only one import line added to the head of the web page, it allows to easily add the neuGRID's dashboard to any web application. The glowing brain shown in the neuGRID menu, representing the Grid load, is dynamically updated using an AJAX query where the server returns a JSON (JavaScript Object Notation) representation of the Grid load. The Grid load is computed on server-side based on the information made available by the neuGRID grid infrastructure.

Usage of the organization/community/user model in the Liferays's portal allows creating a clear structure representing the neuGRID architecture:

- one organisation: neuGRID
- one location/sub organization by site: KI, FBF, VUMc
- one community per project: E-ADNI, AddNeuroMed, etc
- users will belong to the main neuGRID organization and to the corresponding location

Users are able to join and quit communities according to their interests. Joining community is moderated using restricted communities feature. Each community provides social tools to users (a shared calendar, a document library, a WiKi, message boards, a blog etc). It is even possible to have one specific theme per community. Liferays also allows the users to have their own sets of private and public pages, but in order to keep a simple and clear portal this feature has been deactivated for now. One portal will be deployed for the whole infrastructure with portlets allowing querying of the services/facilities, hosted at KI, FBF and VUMc, as per the sites and project policies. If needed WSRP 2.0 portlets could be integrated in the portal, but in order to get as much responsiveness as possible and to ensure an easier administration, local JSR 286 portlets are preferred. The main public part of the site is implemented by creating pages into the default guest community. It presents the neuGRID project and explains how to join neuGRID as a user.

The Liferay-based portal was moved to http://neugrid.healthgrid.org and replaces the old Ruby On Rails-based portal. Despite the fact that the Liferay-based portal was actively developed, there is still some work to be done. The user's communities/groups have to be extracted from the CAS SAML assertion.

The custom CAS logout has to be integrated, perhaps by completing the development of a NeuGRIDFilter class, implementing javax.servlet.Filter.Filter and allowing handling of the Single Sign Out of both Liferay and CAS.

5.4 Implementation Details and Environment

The Liferay version used is the latest Community Edition version 5.2.3 along with the corresponding Plugins SDK. The neuGRID Liferay portal is deployed in Tomcat 6 with a MySQL 5.0 database and is using a Sun Java 6 JDK on a debian Lenny paravirtualized Xen host.

Historically, Liferay has to be extended using the EXT environment, which allows modification of any part of the portal source code, but in the recent versions, the so-called "Plugins SDK" is provided. This Plugins SDK allows development of portlets, themes, layouts, hooks and Web Applications with a lower coupling with the main Liferay portal source code. They are built against the Liferay API and not against the

implementation classes like with the EXT environment, therefore, the neuGRID theme is developed using the Plugins SDK. The neuGRID menu is developed as a portlet using the Plugins SDK. The neuGRID terms of use page customization is made using a JSP hook, allowing customization of any Liferay JSP pages bundled with the portal, without having to use the EXT environment.

The development of the portlets/hooks/themes etc is IDE (Integrated Development Environmet) agnostic. The Liferay team provides the ant scripts required to build and deploy either the whole portal, the EXT environment or plugins for portlets and themes.

5.5 Future Directions

Given the fact that it is quite a complex architecture, the processing of requests could be a bit slower; cache techniques have been developed to address this problem. The technical choices which have been made will also require more work from the service providers as they will have to provide portlets for enabling access to their web services.

Once the portal is ready, it will be moved to a production environment. The customization of its comportments will be a required step to have the best possible user experience. Liferay provides facilities such as clustering, load balancing and advanced caching techniques which will help in ensuring the robustness of this solution. This will likely be achieved before the end of the project.

6. The Anonymization Service

6.1 Introduction

The purpose of the anonymization service is to facilitate the pseudonymization and de-identification of the data that is stored within the neuGRID infrastructure. In order to make the data available to the users for analysis, the anonymity of the patients should be preserved.

Pseudonymization is defined by Wikipedia as "a procedure by which all person-related data within a data record is replaced by one artificial identifier (like a hash value) that maps one-to-one to the person. The artificial pseudonym always allows tracking back of data to its origins which is the difference with anonymised data, where all person-related data that could allow backtracking has been purged." The pseudonymization process includes the checking of files to ensure that all the markers which can provide information to identify a patient are removed before the image can be made available. Most of the time this will be done by removing the image file's text tags containing metadata about the patient such as name, date of birth or any other information that could identify them.

In exceptional cases however, it may be necessary to de-anonymize the data, i.e., identify the subject the data originated from. In order to allow de-anonymization, a key will be generated and stored in the PatientID tag of the files. Following multiple discussions which occurred within the consortium, especially with WP2 colleagues, it was concluded that returning this key to the entity using the service will be sufficient to ensure compliance with regulatory and privacy/confidentiality requirements and ensure an adequate level of protection for data subjects in compliance with the existing European legislation on data protection.

6.2 Architecture Description

There are three main technical components of the anonymization service:

- The neuGRID pseudonymization library
- The Pseudonymization Web Service
- The Pseudonymization applet/stand alone application

The neuGRID pseudonymization library is a Java library providing the necessary methods required to pseudonymize DICOM images in a reusable form. In order to leverage the work and reuse what has already been done the library is using an already existing DICOM manipulation library, the dcm4che2 toolkit. dcm4che2 is a high performance, open source implementation of the DICOM standard. The neuGRID pseudonymization library allows sharing of the pseudonymization code between both the Pseudonymization Web Service and the pseudonymization applet/standalone application ensuring that the same coherent level of pseudonymization is used throughout the neuGRID infrastructure.

The Pseudonymization Web Service is the point of entry in the neuGRID infrastructure to pseudonymize and upload the images. It allows users to send images for pseudonymization and once done, these images can be uploaded on the GRID, registered into LORIS and used by neuGRID users.

The Pseudonymization applet/stand alone application is a tool that will be provided to neuGRID's users to help them pseudonymize images without having to send the images outside the walls of the hospital. The pseudonymization applet will make full usage of the latest Java features allowing the creation of an applet that can be dragged out of a browser and that can be used outside of the browser's context without having to be connected to the Internet.

To provide a web- friendly access, the following two steps have been planned initially:

- The development of an applet for the pseudonymization of images.
- The adaptation of the applet to allow it to be fully functional outside of the browser.

The data anonymization process is described through the steps shown in figure 30 whereas in figure 31 an overview of the Service deployment is shown.

6.3 Functionality and Features

The neuGRID pseudonymization Java library allows one to keep or remove a selected list of fields from DICOM images. The library is being implemented into the *org.healthgrid.neuGRID.pseudonymize.Pseudonymization* class providing the following public methods:

boolean Pseudonymize(String inputFile, String outputFile, int[] headers, int operation)

FileInputStream Pseudonymize(FileInputStream imageStream, int[] headers, int operation) (not yet implemented)

boolean RemoveHeaders(DicomObject dcmObj, int[] headers)

ArrayList<Integer> KeepHeaders(DicomObject dcmObj, int[] headers) (not yet implemented)

The pseudonymization applet will allow users to select one or more local files and pseudonymize them in a manner configured for the neuGRID needs. It will take a list of images as input and will either output pseudonymized images into one chosen repository or send them to the neuGRID infrastructure according to the user's choice. The applet will also be handling the manual upload of files at the DACS level.

Due to its applet nature when new features, bugs fixes or new pseudonymization rules are implemented, they will be immediately available for use. When running in standalone mode the applet will have an update facility allowing it to update itself and to apply automatically any possible new pseudonymization rules.

In order to be able to create a nice and responsive applet that could live outside a browser, the JavaFX SDK has been used. JavaFX is a client platform developed by Sun allowing the creation of rich user interfaces for Internet applications. It is one of the main RIA (Rich Internet Application) frameworks. It allows reusing any existing Java library, and it is also capable of creating applets that could be deployed on a user's desktop just by dragging them from the browser.

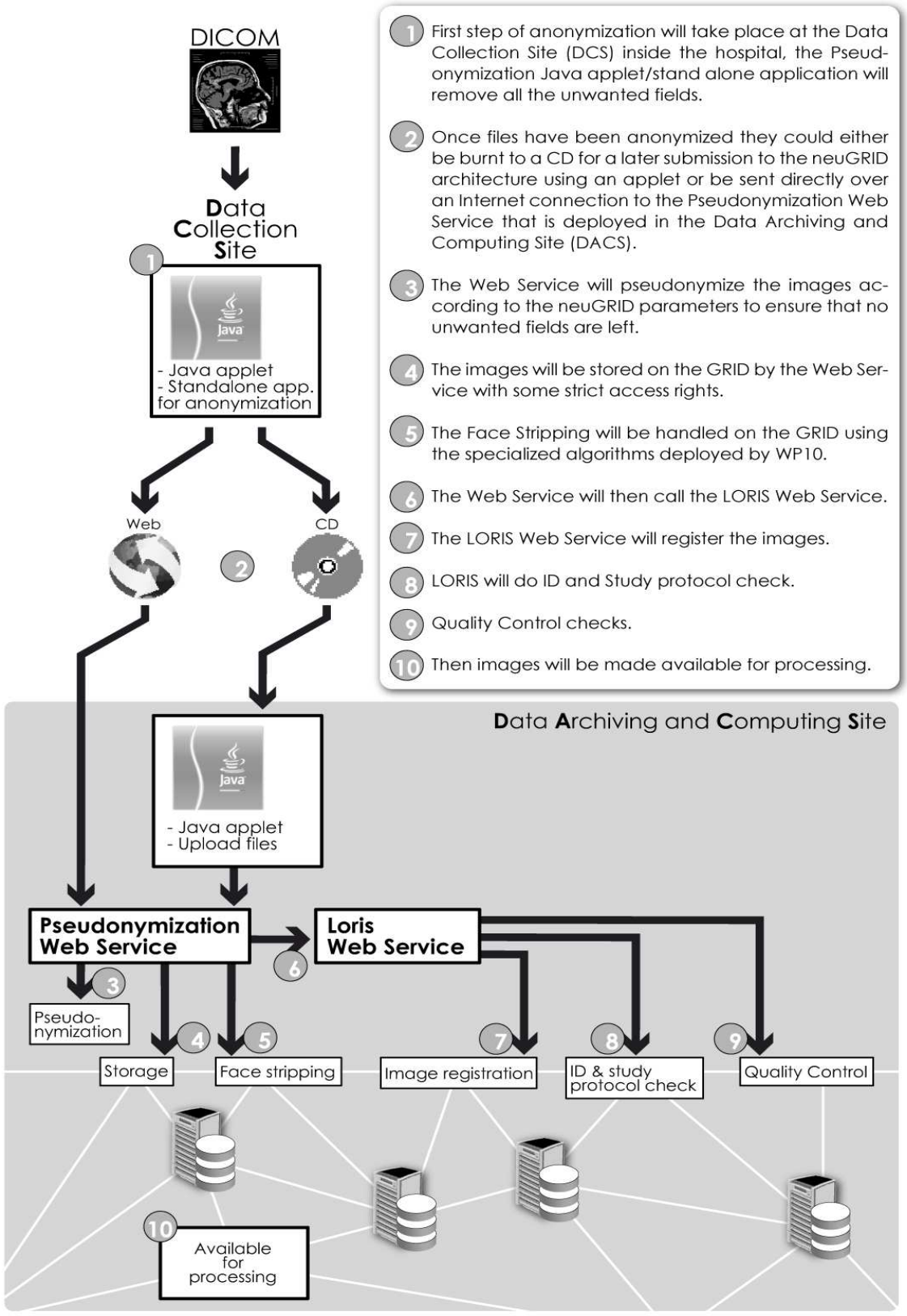


Figure 30: Image upload workflow

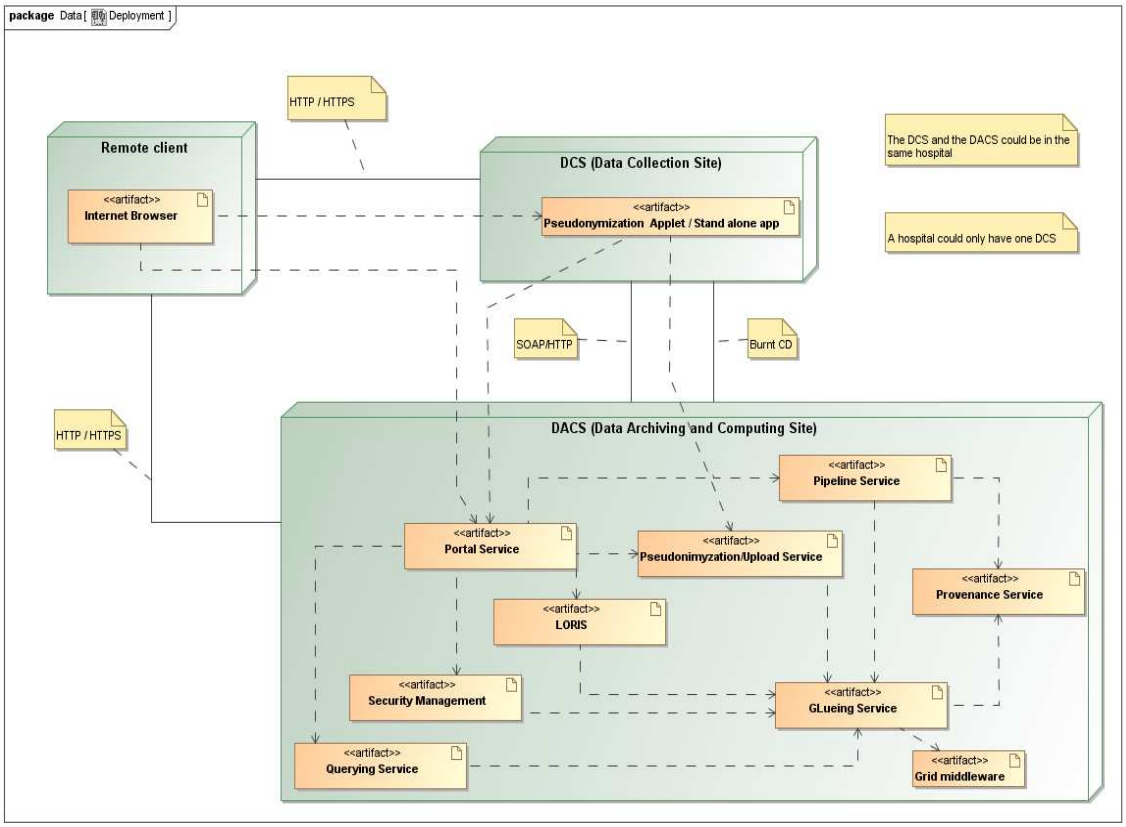


Figure 31: Overview of the Pseudonymization Service deployment

The web service that is responsible for the pseudonymization at the DACS level and for their storage for further treatments receives the DICOM files as a SOAP attachment, in addition to a list of fields/headers, and an operation that could be “keep” or “remove”; an existing neuGRID PatientID may be provided to add images to this very specific patient. To minimize memory consumption on the web service call, the images are saved in a work directory, and the pseudonymization is performed in the working directory.

If the operation is “keep”, the web service will only keep the specified list of headers and drop all the others from the attached DICOM files while ensuring to keep a valid DICOM file by preserving required metadata. If the operation is “remove”, the pseudonymization web service will remove the specified list of headers of the file and keep the others. At the end of this operation the generated neuGRID patient key (a Universal Unique Identifier or UUID), which replaces the real PatientId, will be returned to the client as a part of the operation status.

Once pseudonymized, the images will be uploaded on the GRID, with restrictive ACLs, and the face scrambling will be performed in the GRID. Then the LORIS web service will be called in order to trigger the registration of the images. At any time a user will be able to connect to the web service to get the status of the actual step, and once all the steps are over, the user will get the LFN of the directory containing the files. The whole process is depicted in the sequence diagram shown in figure 32.

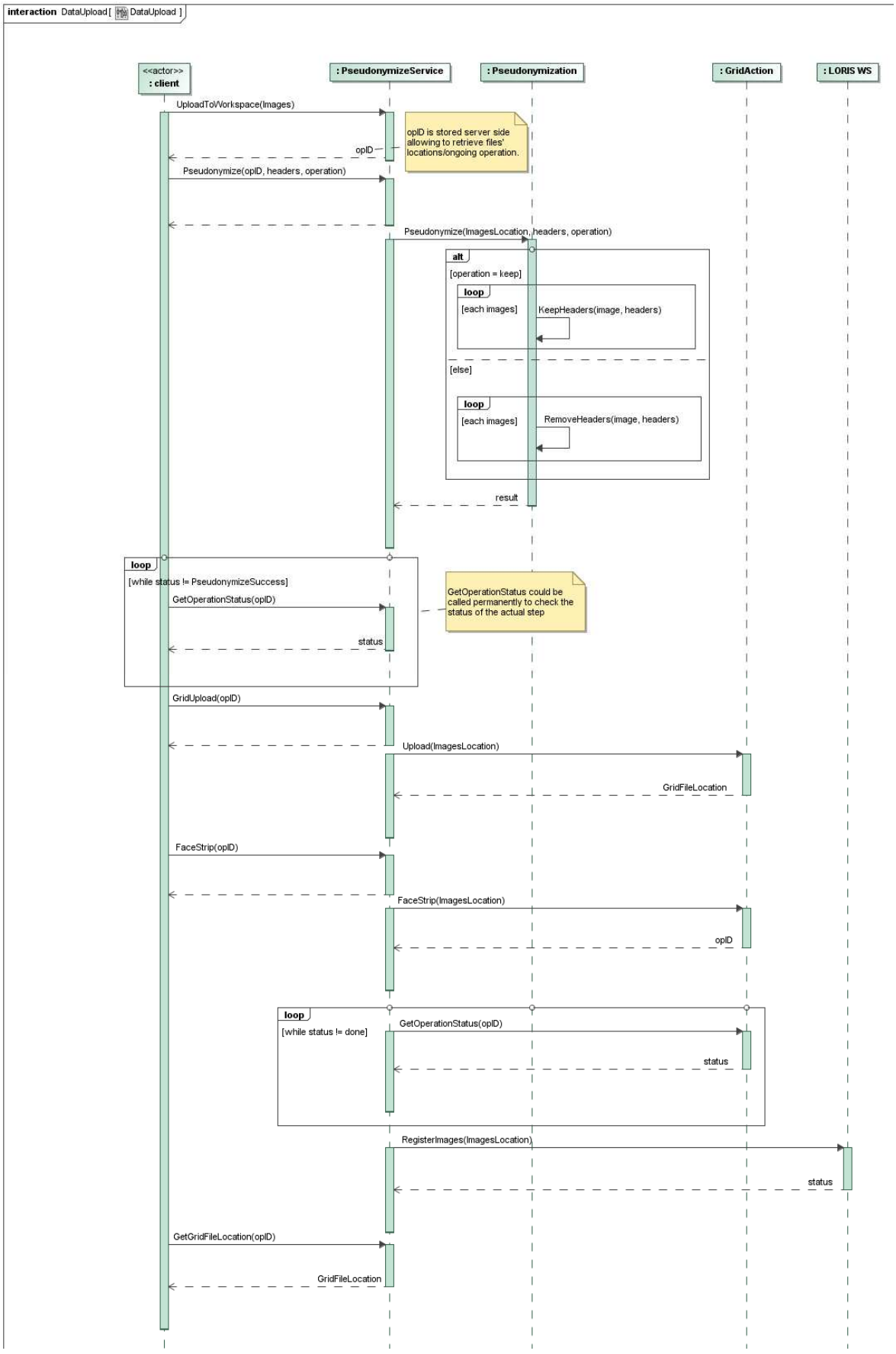


Figure 32: Pseudonymization Service's sequence diagram

6.4 Implementation and Environment

The pseudonymization library is a single Java class making use of the dcm4che2 toolkit version 2.0.20. It is compiled as a jar in order to easily share it between the different actors of the pseudonymization process. A test class based on the jUnit unit testing framework has also been developed to ensure the library performs as intended. The applet is implemented using JavaFX. The JavaFX Script programming language allows using all existing Java libraries like the neuGRID pseudonymization library.

The web service is developed from scratch using a top-down approach by firstly writing the WSDL and secondly generating the stubs for the service using the axis2's wsdl2java command and then implementing the web service interface. The upload process is performed using MTOM attachments. The web service is tested using Axis2 1.4.2 in tomcat 6 using Java 6 on a Xen paravirtualized debian Lenny system. A basic Java client was also developed to test the web service upload operation.

6.5 Issues and limitations

The service is not yet fully implemented:

- For now only the implementation of the removal of a list of selected fields was implemented into the pseudonymization library.
- The applet is in an early stage of development, and many features still have to be added.
- The integration with the Grid environment for uploading the files and running the face stripping is ongoing.
- The authentication and authorization aspects have already been extensively discussed within the consortium, and the implementation should follow in the near future.

Uploading a large number of DICOM images at once could be quite challenging and needs to be heavily tested in order to handle possible upload problems in the best possible way. Moreover, running JavaFX requires a relatively recent version of the Sun Java runtime installed, which is JDK 6 Update 13 on Windows, GNU/Linux and Solaris and JDK 5 Update 16 on Mac OS X. The additional Java FX runtimes will be automatically deployed under GNU/Linux Windows and Mac OS X environments.

6.6 Future directions

The next step is to complete the development of both the Pseudonymization web service and the applet. Once this is done, the web service will be deployed into the neuGRID development environment to ensure the perfect integration with the neuGRID architecture and to be able to exhaustively test the service in a production-like testbed. In order to access the Grid, the Pseudonymization will have to work with the Glueing Service. Integration with the Provenance Service would also be required to record the importation of images into neuGRID.

7. The Way Forward

This section describes the future work that will be carried out in year 3 of neuGRID. In the first two years of the project work has proceeded as scheduled. The user requirements have been elicited and after careful analysis, components have been identified that address particular requirements. The components have been designed, interfaces have been defined and these components have been packaged into services. The nature of services, their roles and the provided functionality has been clearly specified. The design philosophy, which will guide the implementation process, has been laid out and the associated design principles have been described. After the design process, exhaustive investigations were made to evaluate and identify the technologies that can offer the functionality that is required by these components to address the user requirements. A list of the technologies was prepared that can in full or partially implements the desired features in the services and at the same time missing functionality has been highlighted that can not be addressed by the available technologies. By the end of year 1, we had made significant progress towards the WP6 objectives by having a clear roadmap for the services delivery.

In year 2, we built upon the previously highlighted achievements and finalised the services designs. After this, efforts were kick-started for the services implementation and significant progress has been so far made on this front. The Pipeline Service has now sufficient functionality to address the user requirements. It can help users in specifying workflows, transforming the workflows into a common format for wider enactment and can offer the functionality to run these workflows in a distributed environment. The Glueing Service, due to its central and important role to interact with the Grid resources, was the second important service where most of the resources were invested in year 2. It can now offer users the functionality to submit jobs to a Grid, to read, write and browse files from Grid resources and to monitor the jobs. It has been integrated with LORIS. The Provenance Service can capture and store provenance and the querying service will be offered to the users as soon as the provenance data becomes available. Moreover, the Portal Service has grown into a mature service and offers the functionality for a single sign-on as well as providing the CAS support to authenticate and authorise users. The Anonymisation Service has basic support to anonymise the data.

Significant progress has been made towards service design and implementation, however, the ambitious objectives that have been set in WP6 require further major effort that will be undertaken in year 3. We intend to release the services in phases in year 3 where services will be tested, integrated, quality assured and released for user feedback. In year 3, the following activities will be performed to implement the remaining functionality and release the services.

- In the third year, the **Pipeline Service** will be released with the features already described in this document. In addition to this, some additional developments will be made. The Pipeline Service needs to coordinate results retrieval with the Glueing Service and the Provenance Service. Current monitoring information received from the Glueing Service is inadequate for comprehensive provenance. The Glueing Service needs to be extended to gather scheduling information and detailed logs of tasks in addition to the output specified in the JDL. A major component of the Pipeline Service that has yet to be implemented is the workflow planner. The issues raised in the previous section directly have an impact on the workflow planner. Moreover,

for efficient planning, knowledge of the Grid environment is required. The information, such as how many sites are available, which replicas are present and where different tasks have been deployed, is required to efficiently plan a workflow. The Glueing Service needs to get this information from the Grid information services. Currently such functionality is not present in SAGA. Hence in year 3, the issues that have been raised will be addressed and the workflow planner will be implemented to complete the proposed Pipeline Service architecture.

- The **Provenance Service** will require a few changes in CRISTAL and a number of additional developments to offer the features that are required by the users of the project. The CRISTAL structures should understand the structure of neuroimaging pipelines and scientific workflows in general. It should allow users to capture, browse, reconstruct and validate the pipeline related provenance information. The current functionality was not implemented for scientific workflow provenance and therefore this feature needs to be extended to allow an improved workflow support. In neuGRID each user will have separate authentication and authorisation credentials. Therefore the issues such as granularity of authorisation and synchronisation of the CRISTAL data security with the security deployed in the rest of neuGRID need to be further explored. CRISTAL needs to expose suitable interfaces that will allow applications to make use of Grid/Cloud resources through the Glueing service. This approach will not tie CRISTAL down to a particular application or middleware platform. The current schema and database access mechanism needs to be refined to provide a fine grained provenance storage mechanism. CORBA related dependencies need to be removed. The provenance information may be stored on remote databases, which will have to be accessed through SOAP or similar protocols and such support is necessary in CRISTAL. The current provenance reconstruction mechanism in CRISTAL is not sufficient to enable the scientists to reconstruct their workflows. The reconstruction process should help in observing the pipeline creation process, re-executing a pipeline or part of it and modifying a pipeline and storing it with a different version. These developments will take place in year 3.
- The **Glueing Service** exposes SAGA APIs; therefore it can only provide those functions that are supported by SAGA API. The requirements are not fully addressed in the current implementation of the Glueing Service because of the lack of support for those requirements in the current SAGA implementation. As the Pipeline Service generates pipelines or workflows to be executed over the Grid, it needs an enactment engine that can break the workflow into its constituent parts/jobs. It also needs to resolve job dependencies and then execute the sequence of jobs efficiently. The current release of SAGA can only submit one job at a time, through its JavaGAT adaptors, to a submission system such as GridSAM. Thus it does not have support for workload management and scheduling a series/sequence of jobs as per the requirements of workflow. This lack of workflow enactment in SAGA limits the scope of the Glueing Service. Thus, the Pipeline Service that deals with workflows cannot be fully supported by the Glueing Service at the moment. We have developed a temporary workaround for this but it does not support reporting of the workflow status since the gLite adaptor was written for single jobs, though

the gLite middleware can report the status of the entire workflow as a single job. We are currently investigating various possible solutions that will allow the gLite adaptor to report the status of an entire workflow, as well as retrieve the output sandboxes for the jobs that constitute the workflow once they have been executed. We need to implement a single-sign on functionality to facilitate users. Currently to invoke enactment of a workflow and access other resources, the user's certificate, key and associated passphrase are required to initiate a proxy at the Glueing Service end. To use this model the user has to provide all of these details every time a request is made. To cater for this limitation the glueing, pipeline and other services need to be integrated with a Single Sign On service (SSO). This task will also be completed in year 3 when a mature SSO API becomes available.

- One of the important features of the **Querying Service** is to offer the functionality that may enable the users to query the provenance data. This and other features should be available in the release of this service that is expected by the middle of year 3. It became apparent that the Querying Service more than any other, depended on the designs and implementations of other services. The development schedule for this service was therefore re-aligned with the completion of the user requirements analysis as well as the availability of the provenance data. This will allow the querying service architecture to be informed by the requirements of users and provide enough time for other services to reach a level of maturity before any final decisions are taken. In the coming months the query service architecture will be implemented to allow flexible access to neuGRID data. An API will also be developed forming a standard interface to the querying service. This process is timed so that the resulting querying service can be fully tested before it is integrated within the neuGRID infrastructure.
- The **Portal Service** already offers a number of features as has been stated in this document and is available to the project users. The service is ready to be integrated and is waiting for portlet implementations from the service providers. The technical choices that have been made will require more work from the service providers as they will have to provide portlets for enabling access to their web services. The processing of requests is a bit slower in the portal service and improvements are required to address this issue. Cache techniques have been developed to address this problem but more alternatives are being investigated. Once the portal service is ready, it will be moved to a production environment. The customisation of its components will be a required step to have a best possible user experience. Liferay provides facilities such as clustering, load balancing and advanced caching techniques which will help in ensuring the robustness of this service. This will likely be achieved in the third year of the project.
- The **Anonymisation Service** is not yet fully implemented and will require a significant effort in the third year to implement the remaining features. For now only the removal of a list of selected fields has been implemented into a pseudonymisation library. The applet is in an early stage of development, and a lot of features still need to be added. The integration with the Grid environment for uploading the files and running the face stripping is an ongoing process and will be implemented in year 3. The authentication and authorisation aspects have already been extensively discussed within the

consortium, and the implementation should be available in year 3. Uploading a large number of DICOM images at once could be quite challenging and needs to be thoroughly tested in order to handle possible upload problems in the best possible way. The next step is to complete the development of both the Pseudonymisation web service and the applet. Once this has been completed, the web service will be deployed into the neuGRID development environment to ensure the perfect integration with the neuGRID architecture and to be able to exhaustively test the service in a production-like testbed. In order to access the Grid, the Pseudonymisation Service will have to work with the Glueing Service. Integration with the Provenance Service would also be required to record the importation of images into neuGRID. These investigation and developments will be carried out in year 3.

As the requirements are quite clear and all the technological choices have been made, it is expected that the work will be completed within the time frame. It is also anticipated that some additional manpower will be available in year 3 and as a consequence, the speed of progress should increase and better quality implementations of the services will be made available.