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: D6.3 Interim service prototype report

Due date of deliverable: Month 36

Actual submission date: February 7th 2011

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

Executive Summary	**3**
1. The Pipeline Service	**6**
2. The Provenance Service	**10**
3. The Glueing Service	**14**
4. The Querying Service	**20**
5. The Portal Service	**28**
6. The Pseudonymisation Service	**29**
7. Conclusions	**30**
Appendices: XML Schemas for Workflow Provenance	**32**

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 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 has provided an Application Programming Interface (API) which is independent both of the application domain and of the underlying Grid infrastructure.

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 stability, eventual composition into biomedical applications and future expandability. The WP6 deliverable document in year 1 outlined the design philosophy that was followed during the construction of the distributed medical services. The deliverable also described a design for the set of services that constituted the distributed medical services layer. The design and evaluation process was led by user requirements, which have been separately elicited in the project's User Requirements Specification.

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 exist as autonomous and loosely coupled entities that can be executed independently. Each service addresses the requirements that cannot be handled by any other single component or service. Collectively and in cooperation with each other, services support the user analysis process and therefore deliver a functional neuGRID system.

A group of services has been designed and developed that is neither middleware nor application specific or dependent. 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 deliverables D6.1 and D6.2. 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 have been provided to enable interaction with other services and applications.

In year 3, a significant effort has been invested to finalise the implementation of the services. In year 3, most of the activities in WP6 were centred on service integration with the rest of the components in the project. This has included the consolidation of

the functionality, provision of the missing features as a consequence of user feedback, quality assurance and documentation of the services that have been implemented. As a consequence of these developments, a prototype of the services was demonstrated in the neuGRID meeting in Geneva in December 2011. The prototype, by making use of the WP6 services, demonstrated a complete analysis process. The services were shown to deliver the required functionality according to the user requirements specification.

The salient activities in WP6 in year 3 included the following:

1. The integration of the Pipeline Service with the Pandora Integrated Glueing Service has been achieved. The Glueing Service recently has been integrated with the neuGRID Single Sign On (SSO) infrastructure. The Pipeline Service was initially integrated with a version of the Glueing Service that ignored user authentication. After the release of the version of the Glueing Service that supports Single Sign On and user authentication, the Pipeline Service was integrated to the latest version.

2. The Provenance Service has recently matured and has been deployed as part of the WP6 Services infrastructure. The Provenance Service is dependent on the workflow information from the Pipeline Service. Therefore, recent work on the Pipeline Service also focused on achieving integration with the Provenance Service. This primarily involved agreeing on the XML schemas for information exchange and mechanisms for mapping the data models used in the Pipeline and the Provenance Services. The Provenance Service has now been integrated with the authentication-enabled Pipeline Service. The XML schemas for workflow descriptions, instances and status updates have been finalized. The descriptions have been provided to specify workflow items within CRISTAL. The functionality has also been implemented in the Provenance Service to enable writing provenance information to a relational database.

3. The Desktop Fusion environment is a user environment which is available from the neuGRID Portal. A set of user interfaces were developed, which enabled the submission of workflows, tracking of status information as well as various workflow steering functionalities. These interfaces are designed to be used by neuGRID users. In addition to this, a GUI has been implemented to facilitate users in interacting with the WP6 services. The GUI is able to display information from the Provenance Service, Loris and UDDI by acting as a front end for the querying service. The GUI is based on the Liferay portal environment and has been deployed within the neuGRID portal. The GUI is able to download files from the neuGRID infrastructure using the Glueing Service. The GUI is also able to display information, by using either SQL queries from the provenance database, or by using the predefined functions available to it from the Querying Service.

4. The Glueing Service is now stable enough to be able to accept and submit workflows in an XML format provided by the Pipeline Service. A user can retrieve the output produced by a workflow by providing a workflow identifier that is generated during the workflow submission. A user is also able to retrieve the real-time status of a workflow in an XML format. Multiple users can use this service and perform grid operations, such as file management and workflow operations, by providing their neuGRID proxies. The Glueing Service can also provide status information for all jobs in a workflow in an XML format. The Provenance Service makes use of this information to track workflows and analyses.

As discussed in the design document and also in the year 2 deliverable, new user requirements may emerge and new sets of features may need to be added in future. 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. It is believed that most of the essential requirements that have been identified in the user requirements analysis have been implemented. The services were demonstrated to the potential user communities and their feedback was sought as a vehicle for service testing, improvements and production quality releases. In the following sections, we describe the functionality and the progress on the individual services that has been realised in year 3.

1. The Pipeline Service

1.1 Purpose and Introduction

The neuGRID generic medical services layer includes numerous components that facilitate the execution of a neuro-imaging 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 WP6 and WP10. Its role 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 grid-enable and enact 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 document is to update stakeholders regarding the final implementation of the Pipeline Service and to highlight its capabilities and how users can interact with this service. The following items were achieved in Year 2 and have been documented in Deliverable D 6.2.

1. Fully defined web-service interface of the Pipeline Service
2. Implemented a translation component that supports translations for both 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.

In Year 3, most of the work that was performed related to addressing the issues that were highlighted in Deliverable D6.2 and achieving end-to-end integration with all WP6 services that interact with the Pipeline Service as well as developing user-facing client utilities.

This document is structured as follows: Section 2 highlights the final architecture of the Pipeline Service and its interactions with other WP6 Services. Section 3 details how the issues specified in D6.2 were addressed. The section 4 describes the integration with the Provenance Service. Finally, section 5 discusses the user facing user interfaces.

1.2 Architecture

The components of the Pipeline Service are outlined in Figure 1. The motivations for the architecture are highlighted in Deliverable D6.1. The interaction starts with the authoring of a pipeline, that the user wants to execute on the Grid. Authoring can be done in numerous tools. The LONI Pipeline is a popular neuroimaging pipeline authoring environment that is supported by the Pipeline Service. The architecture of the Pipeline Service is flexible and any suitable authoring environment can be accommodated. In Year 3 the support for script based workflows was added.

After authoring the pipeline, the user invokes its submission. In this case, several things happen: firstly the authored pipeline that is represented in a format which is native 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. 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 was presented in the deliverable D6.2. After workflow translation, architecturally the workflow should be planned for the execution. Workflow planning is not part of the final implementation

of the Pipeline Service. The lack of workflow planning is due to the limitations of the underlying engine the Glueing Service uses to submit workflows and management them.

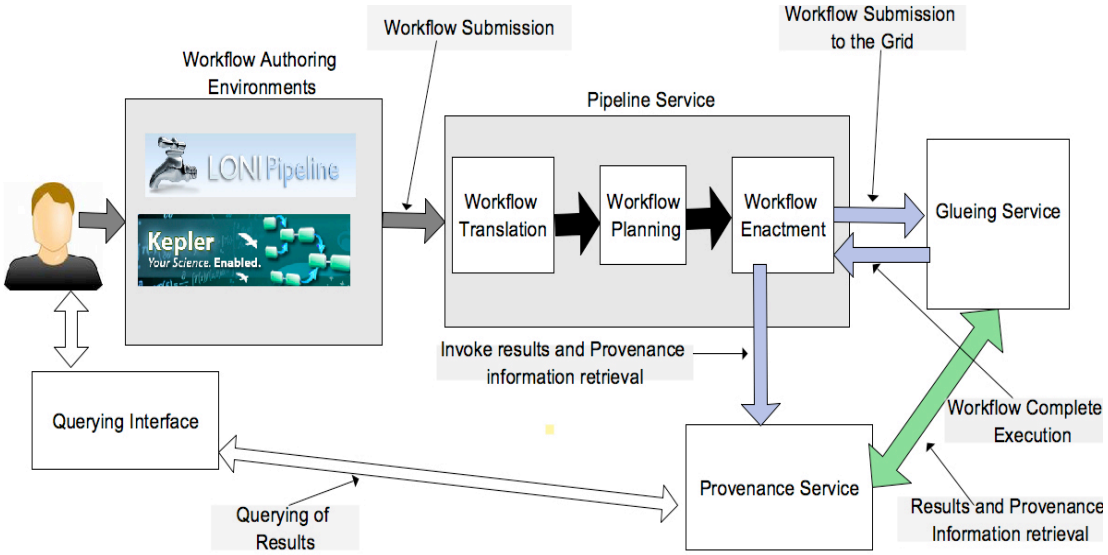


Figure 1: Pipeline Service Architecture

Workflow enactment is the final stage in the Pipeline Service. The Pipeline Service submits the workflow to the Grid via the Glueing Service. All interactions with the underlying Grid middleware are carried out through the Glueing Service. 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 a querying interface that will be part of the neuGRID Portal.

1.3 Addressing the Issues highlighted in Deliverable D 6.2

The following issues were highlighted in Deliverable D 6.2, which were to be addressed in Year 3.

1.3.1 Glite Specific Considerations

The following issue was specified in D6.2:

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

The adaptor issue was addressed by enriching the information provided by the Glueing Service by using gLite-UI mechanisms. For instance, if the JavaGAT adaptor is used to submit a workflow to the Grid environment, it retrieves all output files without retrieving the corresponding logs of the activity. In order to address this problem, the Glueing Service uses glite-UI interface glite-wms-job-output to retrieve

the output of the workflow. The glite-UI interface provides all logging information for each individual job as well as the output.

For fine-grained monitoring information, the glite-UI and glite-wms-job-status are used to complement the information provided by the JavaGAT adaptor. Once the JavaGAT adaptor has instantiated the execution of the workflow, the gLite specific JobID of the tasks and the workflow are returned to the Pipeline Service. The Pipeline Service invokes a special method to retrieve fine-grained status information directly from gLite. This method is not compliant with the SAGA standard and therefore the Pipeline Service does not use the UWESOAPAdaptor to interact with the Glueing Service. The SAGA standard is constantly evolving and has been used whenever it supported the required functionality. For instance, the support for data management operations is complete in SAGA, while support for workflows is completely lacking. Therefore in order to effectively perform workflow management, either the JavaGAT engine was used or gLite-specific interfaces were deployed to complement missing functionality in JavaGAT.

1.3.2 Integration with Single Sign On

Another issue highlighted in D6.2 was the following: Currently to invoke an enactment of a pipeline 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 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.

This limitation has been addressed by retrieving user proxies from the neuGRID Portal environment. For instance, a subsequent section documents the user interfaces for the Pipeline Service. These interfaces are designed to be used by the users from the Desktop Fusion environment available from the neuGRID Portal. The Desktop Fusion environment has access to the full grid proxy of the user. The Pipeline Service client interfaces are designed to forward this proxy to the Pipeline Service, which then forwards it to the Glueing Service. The Glueing Service uses the proxy to perform operations on behalf of the users. This process is more secure as the user no longer needs to share their certificates and the private key or their passphrase.

1.3.3 Integration with the Provenance Service

A major work that was performed in Year 3 was the integration with the Provenance Service. In this section we will explain how this integration was achieved. Figure 2 explains the interaction between the services.

At first when a user request is received (1), the Pipeline Service extracts the description level provenance from the workflow and forwards it to the Provenance Service (2). The description of the schema that defines the description level provenance is provided in the Provenance service section of this document. The next step involves the instantiation of the workflow. Once a workflow is instantiated the instance level provenance description is forwarded to the Provenance Service(3). The description level provenance includes information related to the workflow template. The instance level provenance includes information related to the actual instance of the workflow template. The actual instance of the workflow template includes information such as which data set will be computed by this workflow instance. The workflow template on the other hand includes information related only to the sequence of computations and data flows in the workflow.

Once a workflow is ready for submission, instance level provenance is sent to the Provenance Service (3). The Pipeline Service then submits the workflow to the Glueing Service (4). The Glueing Service returns infrastructure specific information about the workflow. Information such as the gLite specific Job ID is included in this. The Pipeline service then instantiates a demon process (5). The role of the demon process is to retrieve the status of the workflow on a continual basis until the workflow has completed execution. When status updates are received the status information is encoded into the relevant Status schema and dispatched to the Provenance Service(6). Once the workflow has completed execution, then the final update is sent to the Provenance service and the demon process terminates.

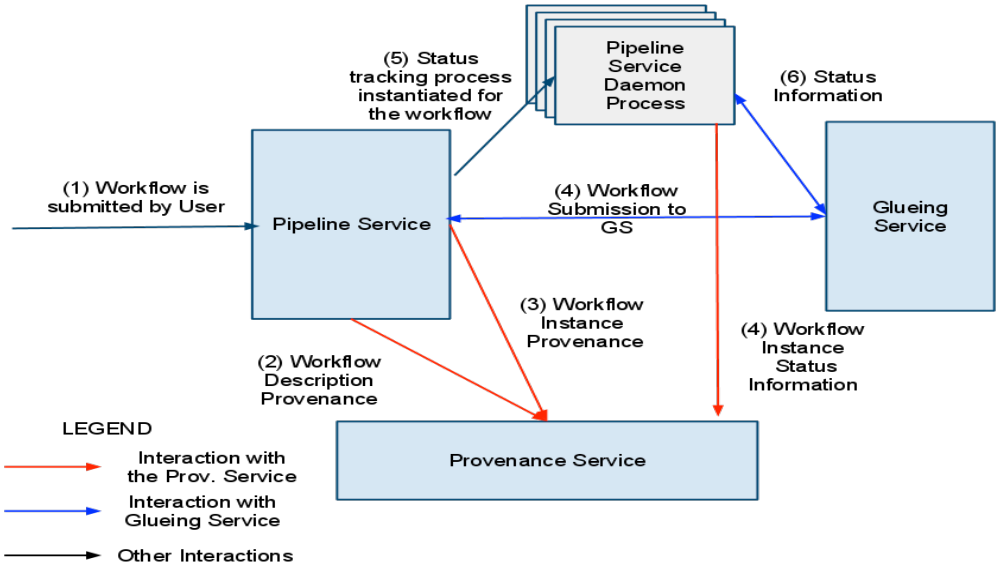


Figure 2: PS/GS and ProvS Interaction

1.3.4 Client Interfaces to the Pipeline Service

User facing interfaces have been developed for the Pipeline Service in the last year. These interfaces enable the user to perform the following actions.

1. Submit a workflow (both LONI and script based workflows are supported).
2. Track status of submitted workflows.
3. Cancel currently executing workflows.
4. Retrieve the output of the workflow, in case of script-based workflows only.

The client interfaces are designed to be used from the Desktop Fusion environment. They forward the current user grid context to the Pipeline Service which is then forwarded to the Glueing Service to achieve credential delegation.

2. The Provenance Service

2.1 Introduction

The need and specification for a provenance service in neuGRID was established in D6.1. To summarise, the Provenance Service is a workflow and data history tracking service. It is designed to capture:

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

The Provenance Service architecture was designed and implemented during the last year. It employs a CRISTAL-based architecture to achieve its goals.

2.2 Architecture

The Provenance Service consists of two layers; the API layer and the CRISTAL layer. Figure 3 shows the relationship between the various components. The individual layers are described below:

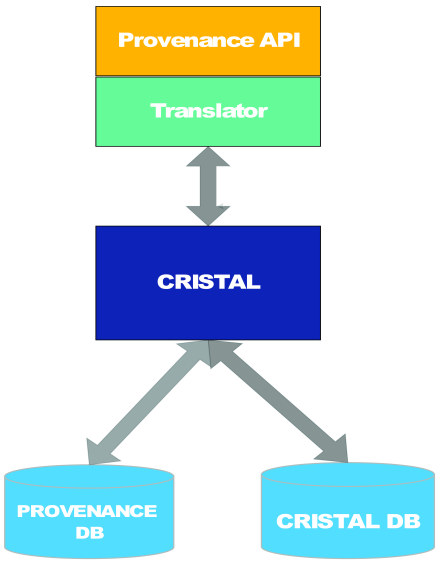


Figure 3: Provenance Service Architecture

2.2.1 Provenance API

This layer implements the Web Service API that serves as entry point into the Provenance Service. It is implemented using Axis 1.4 [http://ws.apache.org/axis/] and Tomcat 6.2.20 [http://tomcat.apache.org/]. This layer also consists of a Translator component. The Provenance API allows clients to:

- Store workflow templates.
- Create workflow instances.
- Update workflow instance status.

The methods that form the Provenance API are listed below. An example use case for the Provenance Service is presented in section 3.

- void storeWorkflowDefinition(String workflowDefinition, String version) throws NoSuccessException

  Store a new workflow template definition in the Provenance Store.

  workflowDefinition workflow definition to store, with workflow ID embedded. See A1.1.

  version version number of workflow definition

- String createWorkflowInstance(String instanceInfo) throws NoSuccessException

  Create a new workflow instance within CRISTAL from an existing workflow template.

  instanceInfo XML-formatted instantiation information. See A1.2.

  return workflow instance ID

- void updateStatus(String workflowInstanceID, String statusInfo) throws NoSuccessException

  Update the status of an existing workflow instance within CRISTAL.

  workflowInstanceID workflow instance ID identifying the workflow instance

  statusInfo XML-formatted update information. See A1.2.

2.2.2 Translator

The Translator is responsible for converting the workflow passed to the Provenance Service in Standard Format (see Secton A1.1) into CRISTAL's own format. The translation is made necessary by the fact that CRISTAL stores workflows that conform to the Control-Flow pattern, whereas the Standard Format workflow follows the Data-Flow pattern. The Translator consists of only one public method:

public CompositeActivityDef translate(WorkflowDesc wfDesc)

where

wfDesc is the workflow in Standard Format.

CompositeActivityDef is the workflow in CRISTAL format.

It employs a 2-pass translation mechanism. In the first pass, the workflow is mined for information about the each activity such as *TaskName*, *Executable*, *Priority* etc. In the second pass, the CRISTAL workflow is constructed using information mined during the first pass. The following rules are followed for each activity when constructing the CRISTAL workflow:

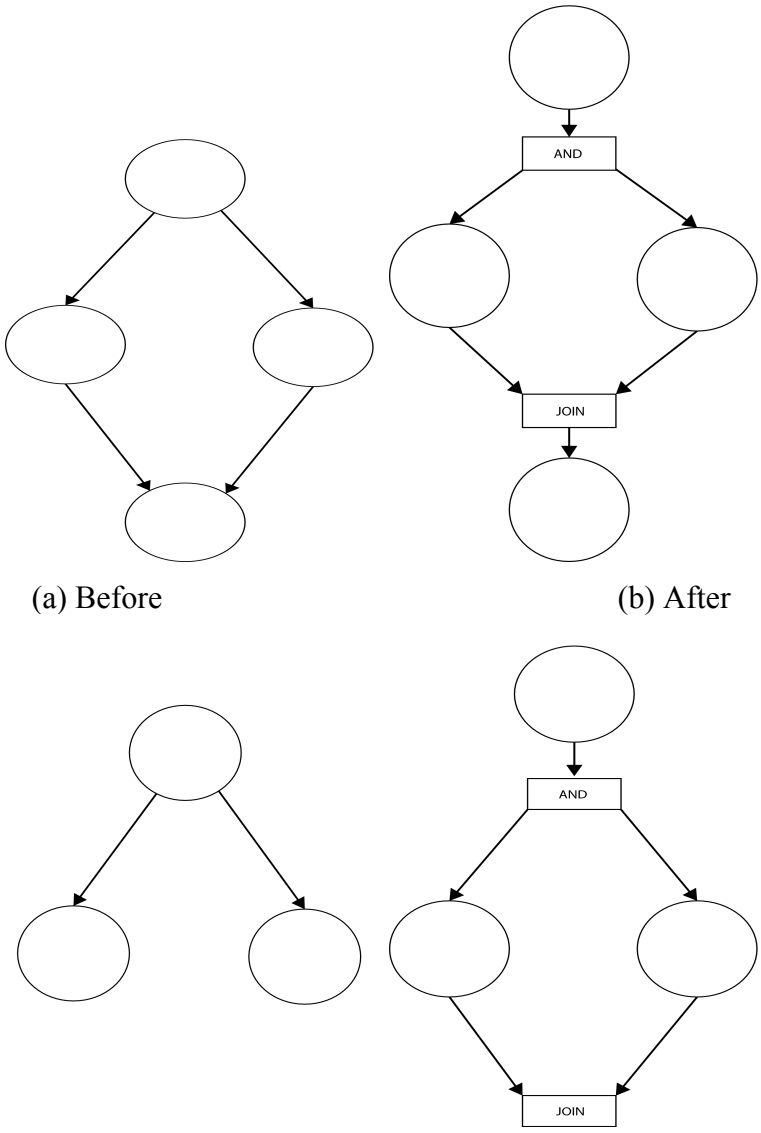
- All activities are mapped using one-to-one mapping into the CRISTAL workflow.
- If current activity has multiple successor activities, a succeeding AND Split is inserted into the CRISTAL workflow.
- If current activity has multiple predecessor activities, a preceding JOIN is inserted into the CRISTAL workflow.

- If current activity has multiple successor activities, all successor activities are connecting to the succeeding AND Split of the current activity in the CRISTAL workflow.
- If current activity has multiple predecessor activities, all predecessor activities are connected to the preceding JOIN of the current activity in the CRISTAL workflow.

Once the workflow has been created, two additional steps are performed:

- If workflow has multiple starting activities, an AND Split preceding all starting activities is inserted in the CRISTAL workflow.
- If workflow has multiple ending activities, a JOIN succeeding all ending activities is inserted in the CRISTAL workflow.

The above two steps are required for creating a correct CRISTAL workflow. Figure 4 shows typical workflows before and after translation.



(a) Before (b) After

(c) Before (d) After

Figure 4: Typical workflows before and after translation

2.2.3 CRISTAL

CRISTAL is the main provenance tracking engine in the Provenance Service. It was developed by UWE at CERN for the tracking of workflows and data of the construction of CMS ECAL. Since CRISTAL is an indigenous product of UWE, expertise for adapting and maintaining it are locally available. Therefore, we decided to use CRISTAL in the Provenance Service.

Once a workflow execution starts in the neuGRID infrastructure, a parallel workflow simulation is created within CRISTAL. This allows clients to send incremental updates to the Provenance Service. The virtual workflow within CRISTAL simulates the actual execution of the workflow on the grid infrastructure. Adapting CRISTAL for the Provenance Service involved creating the appropriate Item descriptions and factories within CRISTAL.

2.3 Sample Program

The following example illustrates how the Provenance Service is intended to be used. The first step is to store a workflow definition in the Provenance Service for instantiation. The workflow is passed to the Provenance Service as an XML string conforming to the workflow definition schema (A1.1). Once the template has been stored, the workflow can be instantiated. To do so, the client must pass the workflow ID along with some instantiation info to the Provenance Service. The instantiation info is an XML string formatted according to the instantiation schema (A1.2). The Provenance Service returns an instance ID to the client that can then be used to update the status of the workflow. To do so, another XML string needs to be passed to the Provenance Service conforming to the status update schema (A1.2). This process is illustrated in the example below.

```
// Store new workflow definition and create workflow instance
storeWorkflowDefinition(wfDef); // Only required when storing new
                                // workflow definition
String instanceID = createWorkflowInstance(wfID,instanceInfo);
updateStatus(instanceID,statusInfo);
```

3. The Glueing Service

3.1 Introduction

Many high-level distributed services such as scheduling, querying, and provenance have been developed in medical domain. Most of these services are developed for a particular community of medical users. As these services are designed and developed by focusing on the requirements of a single community, they cannot be used by other communities due to architecture, interface or platform limitations. The Glueing Service addresses the aforementioned issues by proposing an architecture which shields the heterogeneity of distributed resources and exposes single interface to access those resources. The Glueing Service aims to provide the following.

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.

3.2 Architecture

The Glueing Service is designed to expose SAGA API. The SAGA is an open source standard defined and maintained by the Open Grid Forum (OGF), which describes a high-level uniform specifications to programme applications in middleware agnostic way. The architecture of the Glueing Service is given in Figure 5. At this moment, the SAGA specification does not fully support workflows. The SAGA ommunity is working on a project called 'Digedag', which can support workflows, but the jobs in a workflow are submitted individually, which is not an efficient method to enact and manage workflows in a Grid. Curently the Glueing Service uses JavaGAT to provide grid middleware agnostic functionality. The Glueing Service is making use of the gLite adaptor written for JavaGAT to support file and workflow related operations in the gLite infrastructure.

One of the possible clients to the Glueing Service is the Pipeline Service. The client invokes the Glueing Service methods to perform file operations or workflow related tasks. The Glueing Service uses JavaGAT engine and loads the appropriate adaptor, using a configurable property of JavaGAT, to execute the tasks on the underlying grid infrastructure.

3.3 Addressing the issues highlighted in D6.2

In the last deliverable report (D6.2), the Glueing Service was able to support file related operations. Moreover, the following two major issues were indentified as a work in progress: 1) Single-Sign-On 2) Support for Workflow. During the third year, work has been carried out to implement these missing features in the Glueing Service. The following sections discuss the work that has been carried out to implement these features in the Glueing Service.

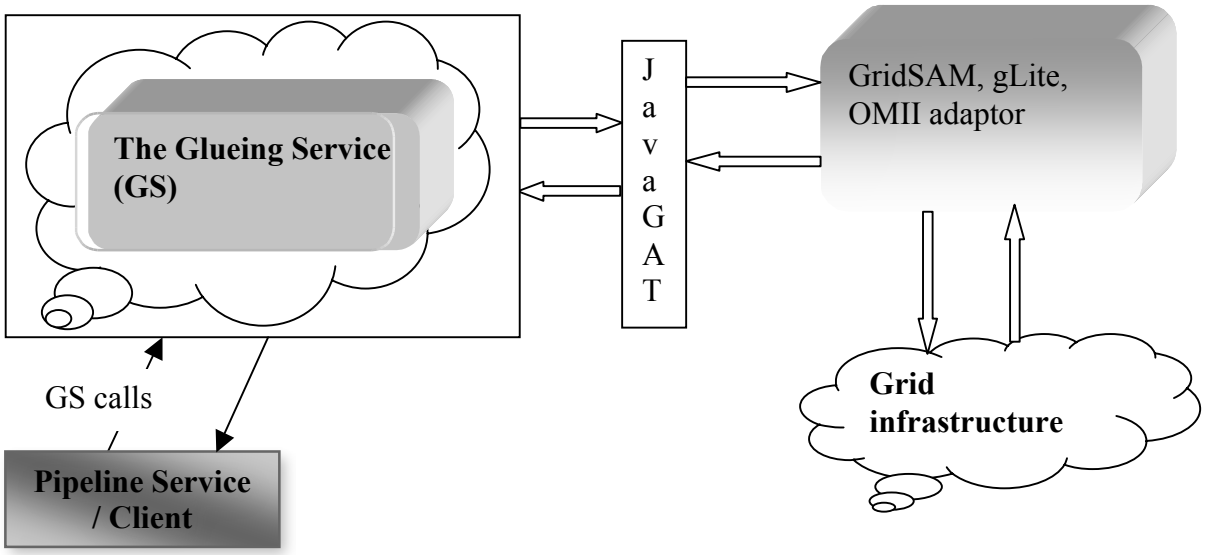


Figure 5: The Glueing Service (GS) architecture

3.3.1 Single-Sign-On (SSO)

"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."

There was an issue to support the single sign on feature in Glueing Service. It is an essential feature in order to perform grid operations for neuGRID users with their own credentials. To address this issue, the Glueing Service provides a session creation approach and uses this session information for grid operations on user's behalf. In this approach, a user or a client of the Glueing Service needs to pass on its grid proxy just once to the Glueing Service. The user can get this proxy from his desktop fusion environment if he wants to access the Glueing Service from his terminal. Other services such as the Pipeline Service, with their portlets inside the neuGRID portal environment, can retrieve user proxy from the portal context. After receiving this proxy string, the Glueing Service stores it locally and assigns a unique identifier to it. This unique identifier is termed as *session_id* in the Glueing Service context. The method exposed for this purpose is named as **createSession(proxy_string)**. The user can use this identifier in subsequent requests to the Glueing Service. On the basis of this identifier, the Glueing Service uses the corresponding user proxy certificate and performs the requested operation. With this architecture, the flow of client request to the Glueing Service and its response is shown in the following sequence diagram.

The Glueing Service uses JavaGAT to perform file related operations. For the implemented prototype, a gLite adaptor has been used to test these operations on gLite-based grid infrastructure. The method 'read' expects a session_id, and a LFN from the user. After receiving the LFN in a service call, the Glueing Service invokes gLite adaptor using GSFile, a layer written on top of the JavaGAT, and passes on the LFN to this adaptor to perform the required operation such as read or write. The sequence diagram in figure 6 illustrates this to perform read operation on a file identified by a given LFN.

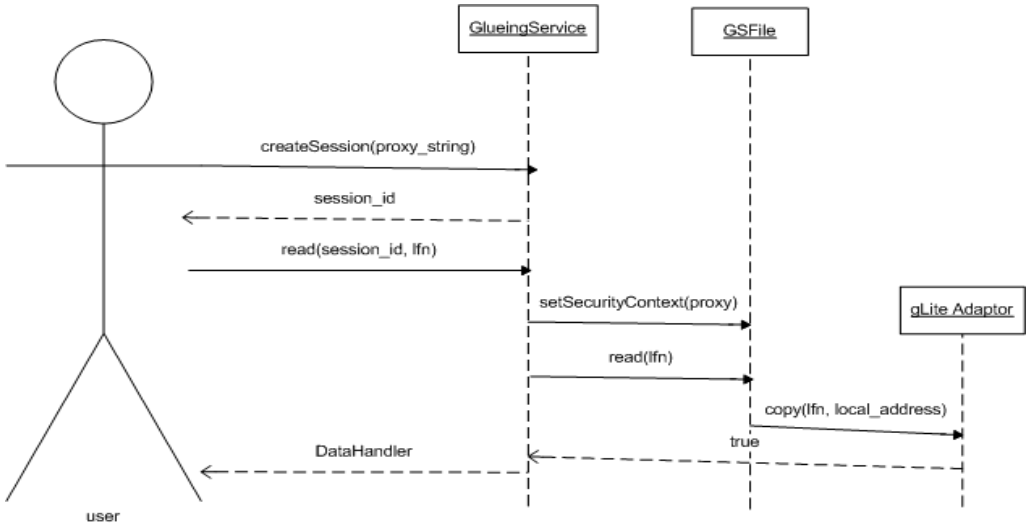


Figure 6: read operation in Glueing Service

3.3.2 Workflow Management

"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. 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."

Now the Glueing Service is capable of supporting workflow operations that a user can use it to submit a workflow and then monitor it. To support workflow related operations, the Glueing Service exposes following web methods.

- String enactWorkflow (String session_id, String workflowDescription, String userCredentials, String pipeline_ID)
- String enactJob (String session_id, String workflowDescription)
- String getWorkflowStatus (String session_id, String workflowID)
- String getJobStatus (String session_id, String workflowJobID)
- String cancelJob (String session_id, String workflowJobID)
- DataHandler getJobOutput (String session_id, String workflowJobID, String pipeline_ID)
- String [] getReplicas (String session_id, String lfn)

The Glueing Service is able to accept and submit workflows in an XML format provided by the Pipeline Service. The Pipeline Service provides an XML representation of a user workflow to the Glueing Service. For this, the Pipeline Service calls **enactWorkflow** method. The Glueing Service submits this workflow using the gLite workflow submission mechanism provided by the gLite adaptor of JavaGAT or through glite-submit-workflow command. In order to use the glite-submit-workflow command, it is necessary to host the Glueing Service on a gLite-UI machine where these commands are available. This method returns a unique identifier called a 'workflowID', which is used to track the status of this submitted workflow. This identifier is returned back to the user to perform the workflow and file related

operations. The required features in terms of workflow management workflow status monitoring, workflow output retrieval and cancelling a submitted workflow.

There is a user requirement where a user wants to submit a single job in the form of a script. This script includes all the information such as environment variables, input files, and output files required to successfully execute this script. A user submits this script to the Pipeline Service. The Pipeline Service transforms this script into an XML that could be submitted to the grid. The Pipeline Service uses a method **enactJob** exposed by the Glueing Service to submit script-based task to the grid.

In order to retrieve the status information of a submitted workflow, the Glueing Service exposes two methods 1) getWorkflowStatus, 2) getJobStatus. The first method i.e. **getWorkflowStatus** provides the status information of an entire workflow. This does not include the status information of the activities within a workflow. The Glueing Service client needs to pass the workflowID, which is created during workflow submission, to this method and it will return status information in an XML format. A sample workflow status XML is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<workflow id="https://wms.maatg.eu:9000/TSOxYTOB8Fl1d_z7mj-Iag" >
   <status>Running</status>
   <status_reason>unavailable</status_reason>
   <Destination>dagman</Destination>
   <submitted>Wed Sep  1 11:26:05 2010 BST</submitted>
</workflow>
```

To get the detailed status information, another method named **getJobStatus** is provided, which returns the status information of each individual activity within the submitted workflow in an XML format. There are four main tags i.e. activity_id, status, status_reason and submitted used in this XML. The *activity_id* is an id assigned to each task by the gLite. The *status* represents the current status of the task. The *status_reason* provides the reason for this status. This information is helpful in case of an error or aborted job. The *status_reason* can help in debugging the error. The 'submitted' tag provides the time of a job submission. This information is helpful for the Pipeline and Provenance services. A sample xml of detailed status information of a workflow is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
 <workflow id="https://wms.maatg.eu:9000/9a8OwrcJdynPWMvUfd6EYw">
  <Job>
     <activity_id> https://wms.maatg.eu:9000/8RXqZyUZncCnsBBCbOkXRw
</activity_id>
    <status>Cleared</status>
    <status_reason>user retrieved output sandbox</status_reason>
    <submitted>Wed Sep  1 16:22:17 2010 BST</submitted>
 </Job>
 <Job>
   <activity_id> https://wms.maatg.eu:9000/Dx72T-
1nTSDkPdsP2I4sag</activity_id>
```

```
    <status>Submitted</status>
    <status_reason> null </status_reason>
    <submitted>Wed Sep  1 16:22:17 2010 BST</submitted>
  </Job>
  <Job>
    <activity_id>
https://wms.maatg.eu:9000/R9NfixqQCsymA9PQQPpa6A</activity_id>
    <status>Cleared</status>
   <status_reason>user retrieved output sandbox</status_reason>
   <submitted>Wed Sep  1 16:22:17 2010 BST</submitted>
  </Job>
</workflow>
```


Once a workflow is completed, the user can retrieve its output by calling **getJobOutput** method of Glueing Service. To perform this operation, the user needs to pass the workflowID and the Glueing Service will return the output in a compressed format such as tar.gz. This compression is performed to reduce the output size and so the transfer time. There could be a scenario where a user wants to cancel a submitted workflow. To support this workflow related operation, the Glueing Service exposes a method called **cancelJob**. A user needs to call this method with the workflowID, and it will cancel user's workflow identified by the workflowID.

In order to plan the workflow activities, the Pipeline Service requires identifying the available replicas of the LFNs provided in the workflow by a user. Based on this information, the Pipeline Service can provide a better execution plan for the workflow activities by selecting an appropriate replica for an LFN. To fulfil this requirement, the Glueing Service provides a method called **getReplicas (session_id, lfn)** which accepts an lfn and returns all its replicas in the form of an array of String.

3.3.3 File Management

Apart from supporting workflow operations, the Glueing Service also exposes methods for file management. Along with other normal file operations such as the size of a file, or owner of a file, the Glueing Service also supports reading and writing files to the grid infrastructure. Reading of an lfn is shown in the Figure 6. To transfer files to the Glueing Service, the axis client uses the Java Activation Framework. In order to write a file to the grid, the Glueing Service receives and writes the user uploaded file to a temporary location. The Glueing Service then loads the specific middleware adaptor based on the site-specific configuration. 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. Some of the other file related methods exposed by the Glueing Service are given below.

- **size (session_id, lfn)**

This method returns the size of a file identified by the given LFN.

- **owner (session_id, lfn)**

This method allows a user to get the name of the owner of this file and who has the owner.

- **listFiles (session_id, lfn)**

This method allows a user to browse a LFN. This will return a list of files available in the given LFN if this LFN is a directory.

- **isLink (session_id, lfn)**

This method checks if the provided LFN is a link. It returns 'true' if the LFN is a link, otherwise 'false' is returned.

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 provides 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. 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 7) provides a choice of ways in which the user can query the data held in neuGRID and has features which include:

- 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.
- A solution which is platform independent and service oriented.
- The creation of a synergy between the querying of heterogeneous data resources and the associated metadata.

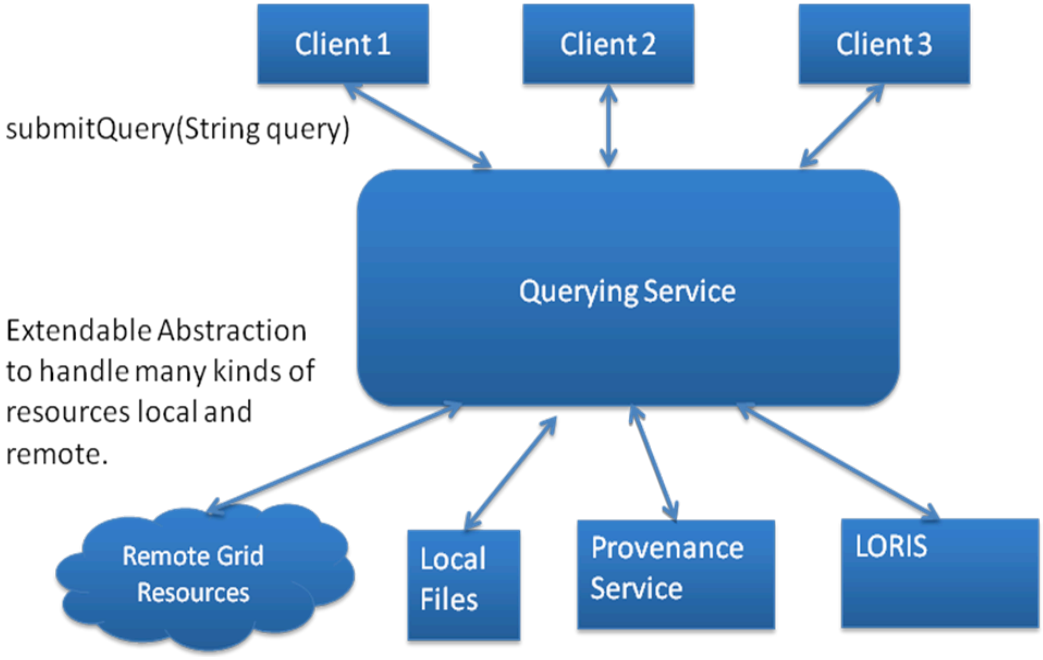


Figure 7: The Querying service will query a number of heterogeneous data sources

The 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. In year 3, a significant activity has taken place on the implementation front and a fairly advanced prototype

of the querying service was shown to the users in the Geneva meeting in December 2010. The following sections describe the work that has been carried out to implement the Querying Service.

4.2 Draft Service Model

The Querying Service (QS) provides a unified abstract access to and seamlessly integrates all data stores in neuGRID including the provenance database, LORIS-X database and UDDI repository. QS acts as a single access point to the neuGRID data hiding the complexity of the implementation of individual data sources. QS does not access the data sources directly (i.e. physical data storage) but uses the data access API (or though a data access service - DAS) of the individual data source which comprises the syntax and semantics of the information stored in that database. The simplified architecture of the QS is presented in Figure 8.

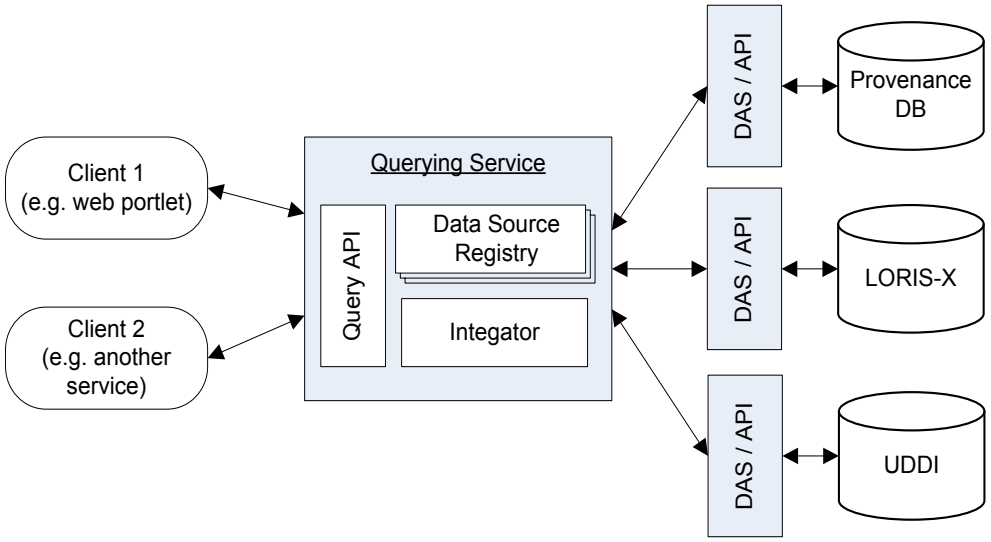


Figure 8: The Querying Service Architecture

All data sources are registered in the Registry which contains the information about the resource endpoint (URL), access method (authenticated or non-authenticated) as well as mappings between the query API methods and those for querying particular data source. When a client sends a query to the service, a particular method is executed which establishes connection to the required data source, invokes the query call, collects the results and sends them back to the client. Authenticated calls are made on behalf of the client: the credentials are extracted and propagated during the call of the respective resource.

The Integrator component is responsible for establishing relationships between diverse data sources to permit querying across different heterogeneous databases. In the current implementation it uses simple mappings between similar attributes in the linked data sources. For instance, Loris-X's *Image.image_uri* field contains the LFN of the DICOM file on the Grid and, thus, can be linked to the input/output data in provenance database allowing concurrent querying of these two databases. Due to the specifics of neuGRID data and data sources we have not identified any further links between resources.

4.2.1 Querying Loris-X

Loris-X database has been designed for the collection, quality control, and maintenance of repositories of medical imaging data and related metadata including

acquisition parameters, image characteristics, attributes of the particular examination, information about the patient etc. (for more information see deliverable D3.2 "Database Implementation and Performance Report") The access to the database is governed by a secured web-service deployed in Pandora container (https://ng-maat-devel1.maat-g.com:8443/lorisx_soap_adaptor/services/Command). The following methods have been implemented (similar to the original functionality provided by the Loris-X):

- *queryTypes* returns the full list of variables/metadata which can be used for writing queries
- *queryForSubjects*, *queryForEvents*, *queryForImages* perform the queries for Subjects, Events and Images respectively preserving the original hierarchy of Loris-X classes: Subject – Event – Examination/ImageFile – Attribute.

In the case of Loris-X, the QS exposes Loris-X query API and mainly forwards clients' requests to the appropriate DAS but in addition it facilitates cross-database queries as explained previously.

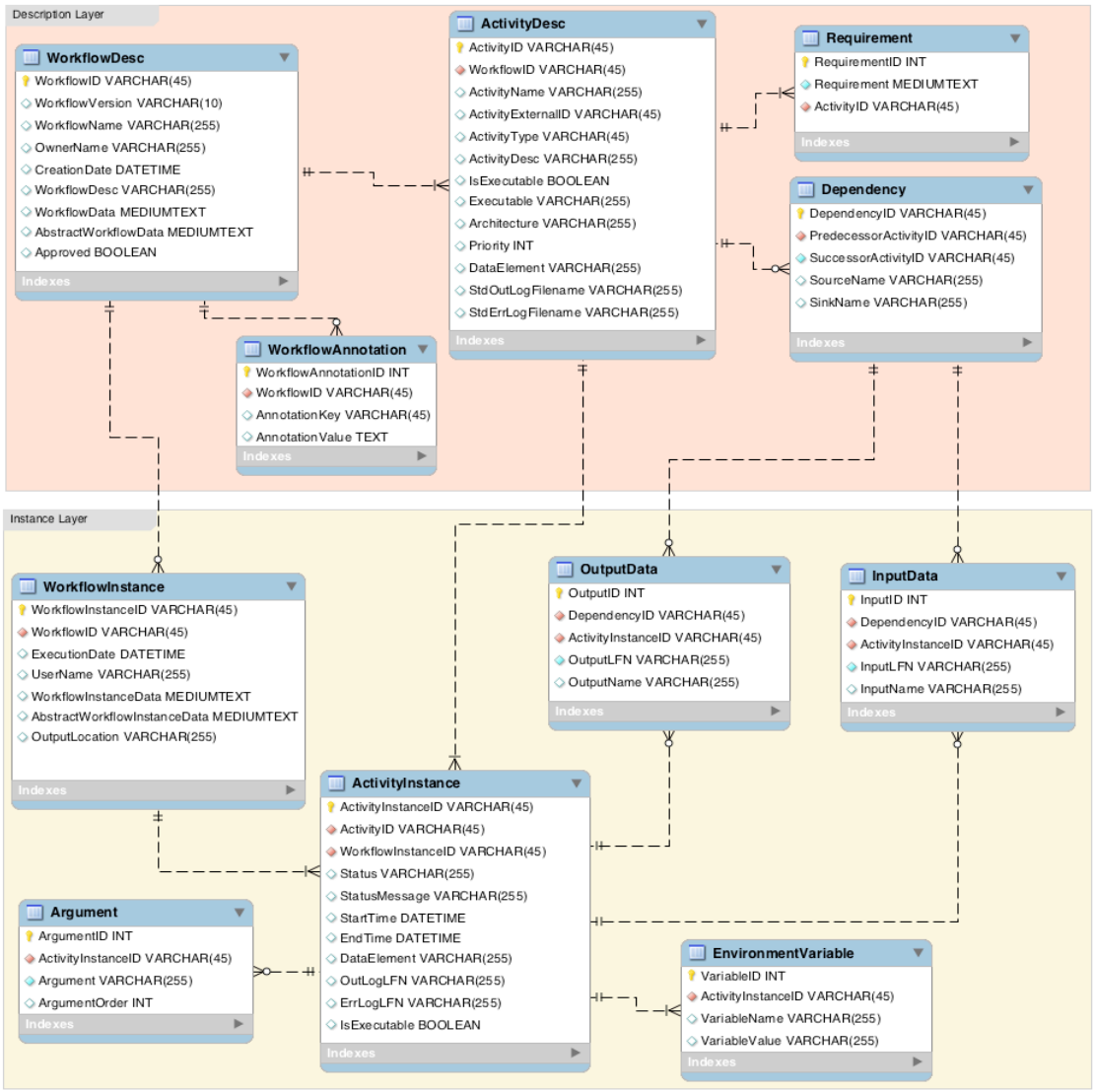


Figure 9: Relational Model of Provenance Database

4.3 Provenance Data Storage and Querying

The Provenance Service stores workflow provenance in the provenance store in order to provide refined data to both clinical researchers and analysis applications. The provenance data store uses a relational database schema and has been designed in a way that it separates the workflow/activity specifications from the workflow/activity instance (execution) related information (as shown in Figure 9). Thus, this enabled rich provenance querying, including queries about the general structure of the workflows, the activities in the workflows, the links between different workflow activities and activity results. Moreover, within the provenance data store any change in a workflow's description is represented by the workflow version. Here, each change in the workflow description spawns one provenance version describing the state of the workflow after the change. Each workflow and its related versions are identified internally by its unique id. In order to identify its source the workflow-id, version-description and times-stamp are utilised. This storage of all workflow versions provides a tracking facility and enables clinical researchers to compare different versions of a workflow.

The *WorkflowDesc* table has *WorkflowID, WorkflowVersion, WorkflowName, OwnerName, CreationDate, WorkflowDesc, WorkflowData*, *AbstractWorkflowData* and *Approved* columns. Here, *WorkflowID* is a unique value assigned to each user generated workflow stored in the provenance database. In addition to the workflow owner related information, this table also stores complete workflow specifications in the XML format. The *Approved* column confirms the approval status of a specific workflow as a boolean i.e. *true/false*. The associated annotations of each workflow are stored in the *WorkflowAnnotation* table that has 1:M relationship with the *WorkflowDesc* table. This *WorkflowAnnotation table stores the information about WorkflowAnnotationID, WorkflowID, AnnotationKey* and *AnnotationValue*.

The *ActivityDesc* table has *ActivityID, WorkflowID, ActivityName, ActivityExternalID, ActivityType, ActivityDesc, IsExecutable, Executable, Architecture, Priority, DataElement, StdOutLogFilename* and *StdErrLogFilename* columns. Here, tbe *ActivityID* column uniquely identifies each activity of a workflow, which is different than the *ActivityExternalID* value. The *ActivityExternalID column* stores the user defined activity identification value. The *StdOutLogFilename* and *StdErrLogFilename* columns contain the file names in which activity logging information is stored. Due to the fact that an activity could be of both *data activity* and *executable activity* type, this information is confirmed by the data stored in the *ActivityType* and *IsExecutable* columns. The Requirement table has *RequirementID*, *Requirement* and *ActivityID* columns. This table stores all requirements associated with a specific workflow activity description and therefore there is a 1:M relationship from *ActivityDesc* table to the *Requirement* table.

The Dependency table has *DependencyID, PredecessorActivityID, SuccessorActivityID, SourceName* and *SinkName* columns. This table stores links to *successor* and *predecessor* activities of an activity. This information is very useful for tracking, editing and recreating workflows. There is a 1:M relationship from *ActivityDesc* table to the *Dependency* table that enables us to query further details of both successor and predecessor activities of an activity and vice versa.

The *WorkflowInstance* table has *WorkflowInstanceID, WorkflowID, ExecutionDate, UserName, WorkflowInstanceData, AbstractWorkflowInstanceData* and *OutputLocation* columns. The *WorkflowInstanceID* uniquely identifies each instance of a workflow. There could be more than one instances of a workflow; therefore, there is a 1:M relationship from *WorkflowDesc* table to the *WorkflowInstance* table. The *AbstractWorkflowInstanceData* column of this table stores complete workflow instance specifications in the XML format.

The *ActivityInstance* table has *ActivityInstanceID, ActivityID, WorkflowInstanceID, Status, StatusMessage, StartTime, EndTime, DataElement, OutLogLFN, ErrLogLFN* and *IsExecuatable* columns. The *ActivityInstanceID* uniquely identifies each instance of a workflow activity. Each activity in a workflow can have one or more than one workflow activity instances; therefore, this has been achieved by having a 1:M relationship from *ActivityDesc* table to the *ActivityInstance* table. In this table the LFNs specified as *OutLogLFN* and *ErrLogLFN* columns contains the logical file names in which the output and error logging information related to the execution of an activity instance are stored.

The *OutputData* table has *OutputID, DependencyID, ActivityInstanceID, OutputLFN* and *OutputName* columns. Similarly the *InputData* table has *InputID, DependencyID, ActivityInstanceID, InputLFN* and *InputName* columns. Due to the fact that there could be multiple inputs and outputs associated with a specific workflow activity, both of the *OutputData* and *InputData* tables have 1:M relationship with the *ActivityInstanse* table. In both of these tables the LFNs specified as *OutputLFN* and *InputLFN* contain the logical file names in which the input and output data of an activity instance are stored.

The *Argument* table has *ArgumentID, ActivityInstanceID, Argument* and *ArgumentOrder* columns. This table has 1:M relationship with the *ActivityInstance* table; and therefore, stores all arguments associated to a specific activity instance. While storing workflow provenance, it is important to store the order of arguments associated with a particular activity instance. Therefore, the order in which the agreements of an activity instance have originally been specified by the end-user is also stored in the *ArgumentOrder* column. Similarly, the information concerning environment variables that are associated with a specific activity instance are stored in the *EnvironmentVariable* table forming a 1:M relationship from *ActivityInstance* table to the *Argument* table.

The following are the brief descriptions of the relations in provenance database:

Title	**Description**
WorkflowDesc	Contains the description of a workflow
ActivityDesc	Contains the description of an activity in a workflow
Requirement	Contains the description of a requirement associated with a workflow activity
Dependency	Describes the dependencies between an activity and

	input/output data
WorkflowInstance	Contains the instances of a workflow
ActivityInstance	Contains the instances of a workflow activity
Argument	Contains the arguments associated with an activity instance
InputData	Contains the input data associated to an activity instance
OutputData	Contains the output data associated to an activity instance

The provenance querying allows a clinical researcher to retrieve information regarding users' specific steps, which can be traced at any point in order to recreate or to clarify the steps taken to produce a result. Moreover, provenance querying can also be used to retrieve information in order to recreate a specific workflow. To enable this, a Provenance Querying Service has been developed, forming a standard web service interface to the neuGRID Querying Service (as shown in Figure 10).

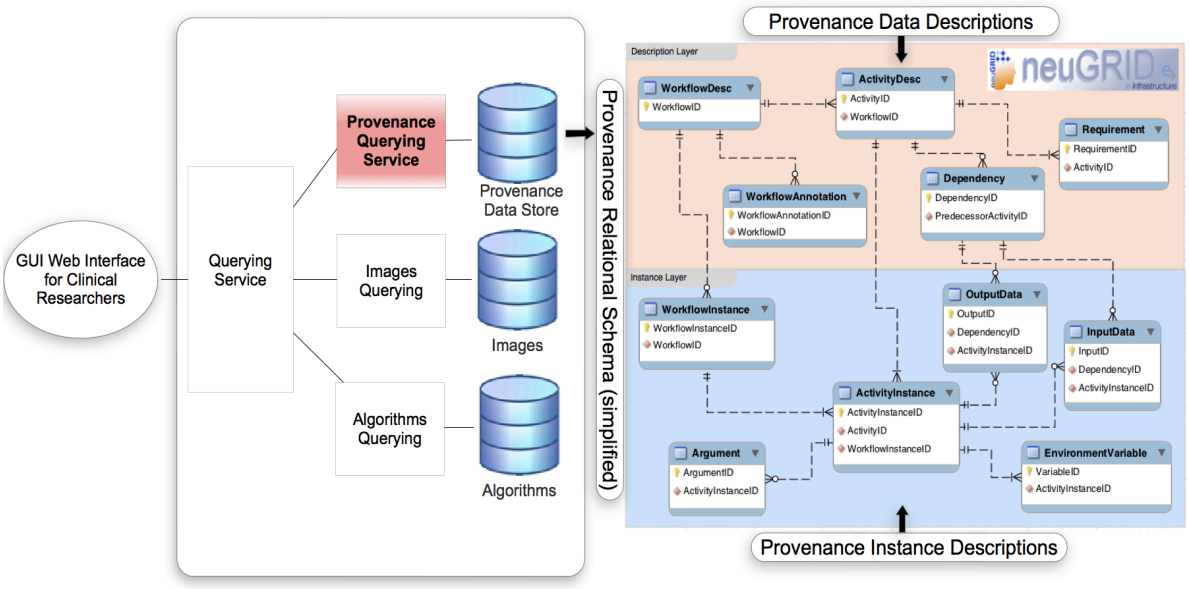


Figure 10: Provenance Querying Service to Retrieve Provenance Data

The Provenance Querying Service has several methods that provide convenient access and enables fine grain searching of provenance data. For example, it supports queries about: the general structure of the workflow, the activities (instances) and the links between activities, the parameters and input data, as well as the execution details and results (e.g. data produced) etc. In this regard, the following are the methods exposed by the Provenance Querying Service.

Service: eu.neuGRID.services.ProvenanceQueryingService

S.No.	Method Name	Description
1	execSqlQuery	Receives an SQL statement from client then (1) executes it; (2) populates java WebRowSet object of the result set; (3) convert WebRowSet into a

		String (XML) object; and finally (4) returns the String (XML) object which at the client side is convertible to webRowSet object.
2	getAbstractWorkflow-InstanceData	Retrieves abstract workflow instance data.
3	getAbstractWorkflowData	Retrieves abstract workflow Data, convertible to java webRowSet object.
4	getActivityInstance	Retrieve the details of a particular activity instance.
5	getArgument	Retrieves the details of a particular workflow activity argument.
6	getInputData	Retrieves the details of a particular workflow activity input.
7	getOutputData	Retrieve the details of a particular activity output.
8	getWorkflowData	Retrieves workflow data.
9	getWorkflowInstanceData	Retrieve workflow instance data.
10	listActivityInstances	Lists the description of all activities in a workflow.
11	listArgument	Lists the description of all arguments of a particular workflow activity instance.
12	listInputData	Lists the description of input data of a workflow activity.
13	listOutputData	Lists the description of output data for a workflow activity.
14	listWorkflowDesc	Lists the description of all available workflows.
15	listWorkflowInstances-Desc	Lists the description of all workflow instances.
16	listWorkflowOfAlgorithm	Lists the description of all workflows in which a particular algorithm is used.
17	listWorkflowOfImage	Lists the description of all workflows in which a particular clinical image is used.
18	writeXMLFromDBQuery	Receives an SQL query from client, and then (1) executes it; (2) populates java WebRowSet object of the result set; (3) write webRS to XML File namely 1.xml and stores it into the bin folder of webserver; (4) convert WebRowSet xml data into a String object; and finally (5) returns a String object (XML), which is convertible to webRowSet object.

4.4 Portal GUI

This section describes the neuGRID portal UI. The purpose of the portal UI is to expose the functionalities of the querying service. The portal UI is based on the liferay portal environment and is deployed fully in the neuGRID portal based environment which is also based on liferay and tomcat. This UI has been programmed in Java using the liferay portal API. Figure 11 shows the UI deployed in the neuGRID portal environment.

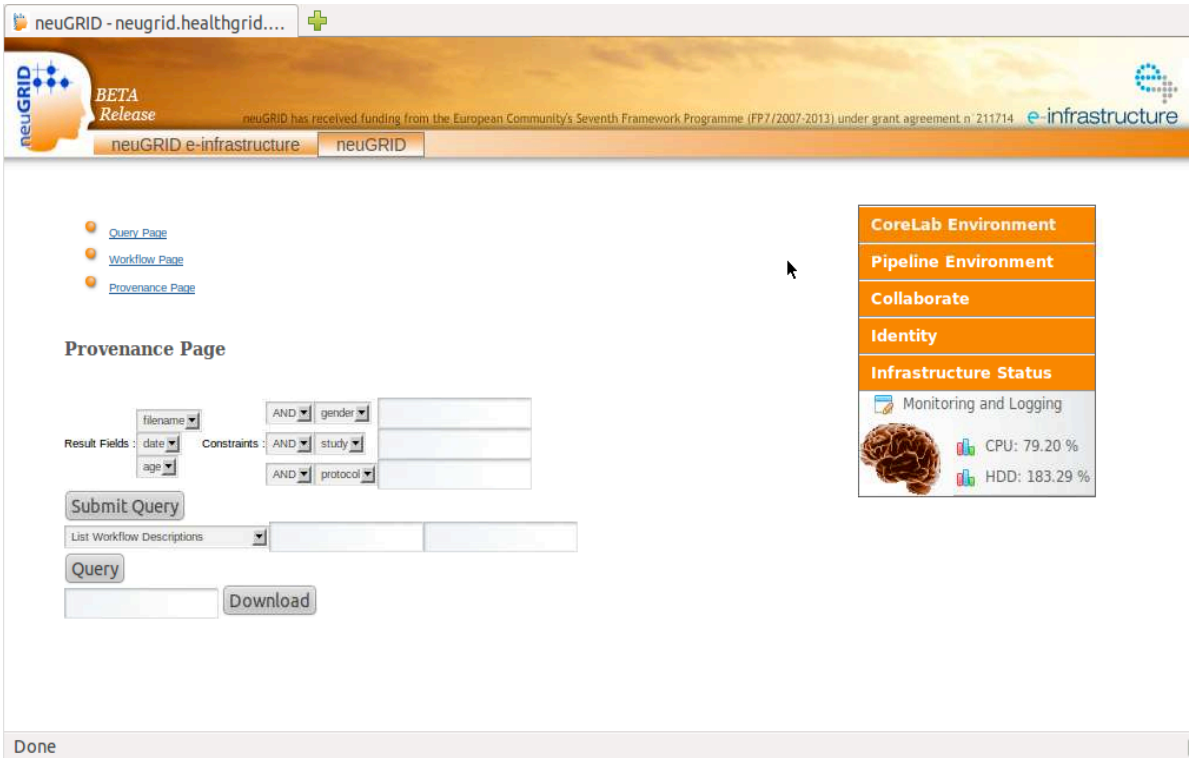


Figure 11: Querying interface in the neuGRID portal

The portal UI is able to expose the predefined functionalities of the querying service. Therefore it can query all provenance information present within the provenance database. The functionality of the UI is currently limited by the functionality exposed via the querying service. It is also possible for users to formulate their own queries via drop down boxes, therefore, extra functionality that is not exposed by the predefined querying service methods can be accessed by a user. This provides a powerful access mechanism for provenance based information.

Another feature of the UI is that it is able to download files from the grid. This is only possible if an LFN is present from a job output. This feature is required so that files can be downloaded directly from the UI, instead of the user having to open another interface to gain access to the grid. The downloading is done by direct interaction with the glueing service. However, for this to be possible a user has to be logged into the neuGRID portal environment with their respective credentials so that access to the grid is possible. This UI can facilitate users in embracing a portal based infrastructure. This will facilitate communication between different applications available from the neuGRID project. All of the neuGRID services have portlet based front ends and can be easily accessible from the neuGRID portal environment.

5. The Portal Service

The portal service is the single point of entry for users to access the neuGRID services. The neuGRID portal has been updated to the latest Liferay version. This was a major upgrade that required a lot of work to be done to be able to profit from the latest Liferay features and capabilities. The neuGRID Single Sign On, the CAS plugin for Liferay and the portal theme were updated due to the major templating changes.

The neuGRID portal's (see figure 12) content is now automatically and completely deployed on the first start up, all the pages are created, the portlets are added to them and the rights are automatically set as needed. The users' accounts are automatically created in the local portal database using the user's attributes retrieved from the SAML assertion that is returned by the CAS server upon a successful authentication.

In order to enforce the collaboration between users, a wiki space allowing trainers and trainees to have a central place to store and exchange documentation, has been added.

Figure 12: A snapshot of the neuGRID Portal Service

6. The Pseudonymisation Service

The Pseudonymisation Service provides an easy way to upload images into the neuGRID infrastructure. It also allows to easily pseudonymise a selection of DICOM files based on some predefined pseudonymisation filters allowing to empty, keep or alter the DICOM fields (see figure 13). These filters are easily customizable/extendable on compile time using a Java class.

The Java FX toolkit on which the applet is based upon has also been updated to the latest version, thus the applet also required to be adapted. This update enforces the compatibility, security and improves the performance of the applet. As Oracle recently announced that the Java FX toolkit's language, the so called JavaFX Script language, has to be replaced by Java , this implies that the applet will have to be redeveloped in the near future. Fortunately, as the applet core is Java-based, only the interface components will have to be replaced, which is already quite a lot of work. We surveyed for a potential replacement and found that the PrivacyGuard Open Source toolkit (http://research.nesc.ac.uk/node/560) for image de-identification project could be a possible choice.

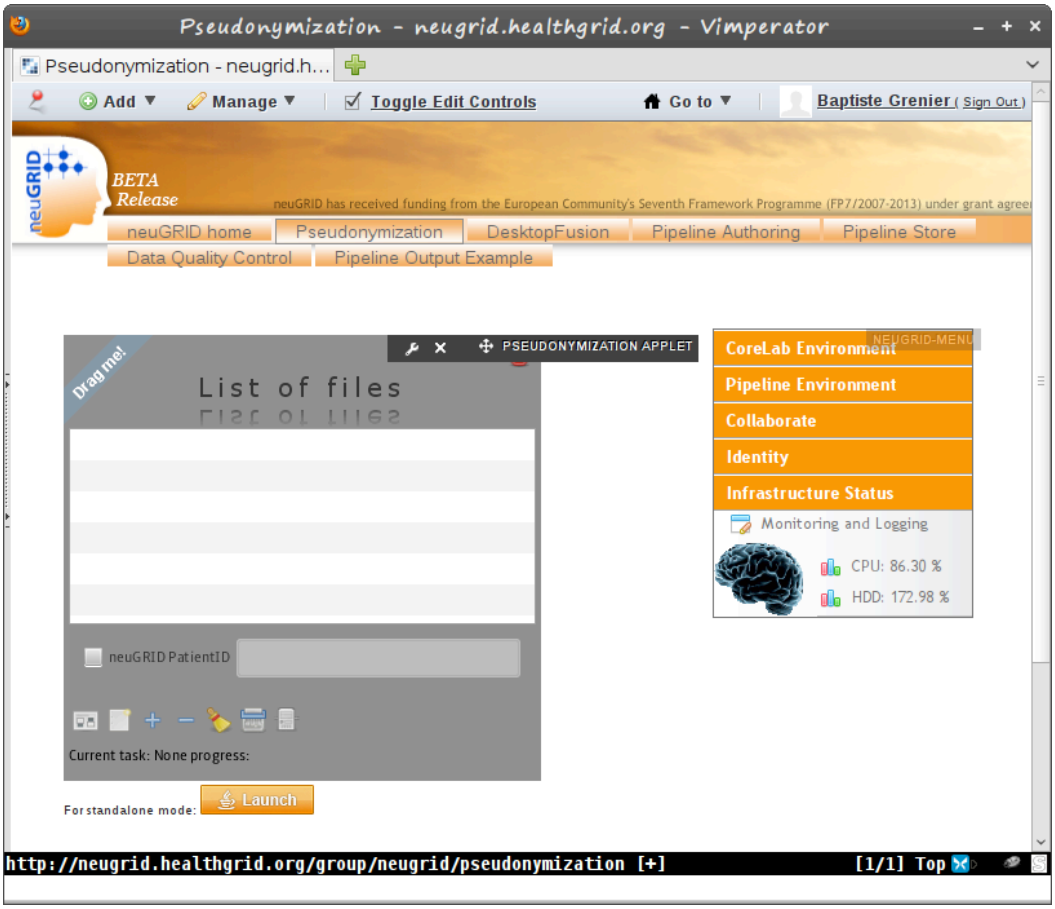


Figure 13: The Pseudonymisation Service interface

7. Conclusions

The aim of the neuGRID project was to provide a user-friendly grid-based e-infrastructure, which should enable the European neuroscience community to carry out research that is necessary for the study of degenerative brain diseases. The WP6 work package, Distributed Medical Services Provision, was contracted to supply a range of services to the users of the project. The creation of a group of generic medical services should enable Grid technologies to be applied in this and a number of other medical domains. This should provide the flexibility that is necessary for interfacing with existing medical systems and should allow the reuse of packaged services that exploit Grid functionality. The aims of WP6 were:

- To produce a set of generic medical services to sit between the user-facing services that are being developed in WP5 and the Grid Services being provided by WP7.
- To provide independence from specific Grid-technologies.
- To provide a flexible and re-usable set of medical services that follow the SHARE recommendations and that can be extendable to other medical domains in the future.

In the design document, an attempt was made to produce designs of the services that can best address the user requirements (as identified in work package WP9) and at the same time are consistent with the design principles that were summarized in the design philosophy. The following sets of activities were performed in order to produce the services designs:

- An analysis of the requirements was carried out to identify the components that can best address the user requirements. A group of generic components that can address these and other requirements that may emerge in future was created.
- The components were identified that can provide common functionality or have overlaps in the functionality they offer. All such components were bundled together to produce a group of components that can address an area of the requirements. The grouped components were exposed as the candidate services that have been described earlier.
- A service design was produced for each of the services using the defined design philosophy. This design process not only glued various components together, it also ensured that they can be extended when required in the future. It was ensured that the minimum dependencies existed between components and services and they were sufficiently scalable to cope with future demands in loads. Particular care was taken to make these services as generic 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.
- An analysis and evaluation process was initiated to investigate emerging technologies and tools that could help in producing the functionality as required in the services. This process helped in identifying a set of technologies that could be reused in implementing the services. This activity also produced a list of functionality in each of the services that could not be implemented using

existing technologies and where new development would have to be carried out. This process also produced a comparative analysis of the state of the art in each of the services and identified missing functionality.

- The WP6 has delivered a set of services as presented in this document. It has also been described in the year 1 and year 2 deliverables how state of the art technologies can implement these services and considered the advantages and disadvantages of each. The year 1 deliveable elaborated the effrots that were made in understanding the requirements, evalauting the technologiees and designing the services. Recommendations were also made regarding the use of interfaces and suitable technologies. The year 2 deliverable summarized the progress that was made towards the implementation of the services. The year 3 deliverable summarizes the efforts that have been made in implementing the missing functionality, integrating the services with the rest of the components and the quality assurance work that was carried out to deliver the services.

The milestones for the year 3 have been successfully completed. The services implementations have been delivered as per the requirements specifications and the designs that were delivered in year 1 and year 2 of the project. The design philosophy was followed during the implementation phase and it was ensured that almost all of the essential features should be implemented. A quality assurance process was followed up and rigorous testing has been made to ensure that the implemented functionality actually works in a real setting. The service components and interfaces were documented and a demonstration of the software was shown to the users in the year 3 quarterly meetings that were held in Brescia, Stockholm and Geneva. We can now say that the project objectives have been satisfactorily addressed as far as the services delivery in WP6 is concerned.

In addition to the services demonstration to the user communities, we also presented the work in the leading international conferences and the software was demonstrated to wider user communities. As per the WP6 objectives, the work should be reused and extended to other applications and infrastructures, therefore an effort will be made to make it useful to communities outside the neuGRID as well as in a follow up project to the neuGRID. The design and implementation of the services already support the provision to be used across applications and infrastructures. To further disseminate the work, we intend to publish the work in leading journals and conferences.

Appendices: XML Schemas for Workflow Provenance

A1.1 Workflow Template

The following workflow template stores a workflow definition in the Provenance Service for instantiation. The workflow is passed to the Provenance Service as an XML string conforming to the workflow definition schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2007 rel. 3 (http://www.altova.com) by Andrew Branson
(CERN) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="WorkflowDesc">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="WorkflowID" type="xs:string"/>
        <xs:element name="Version" type="xs:string"/>
        <xs:element name="WorkflowName" type="xs:string" minOccurs="0"/>
        <xs:element name="WorkflowDescription" type="xs:string" minOccurs="0"/>
        <xs:element name="WorkflowAnnotations" minOccurs="0">
          <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
              <xs:element name="KeyValueSpec">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="key" type="xs:string"/>
                    <xs:element name="value" type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="DataTask" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ActivityID" type="xs:string"/>
              <xs:element name="TaskName" type="xs:string"/>
              <xs:element name="Type" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="namespace" type="xs:string"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="ExecutableTask" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ActivityID" type="xs:string"/>
              <xs:element name="Executable" type="xs:string"/>
              <xs:element name="StdOut" type="xs:string"/>
              <xs:element name="StdErr" type="xs:string"/>
              <xs:element name="Architecture" type="xs:string" minOccurs="0"/>
```

```
              <xs:element name="TaskReqs" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
              <xs:element name="TaskName" type="xs:string" minOccurs="0"/>
              <xs:element name="Type" type="xs:string" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="namespace" type="xs:string"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Dependencies" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ActivityID" type="xs:string"/>
              <xs:element name="Dependency" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="ActivityID" type="xs:string"/>
                    <xs:element name="SinkName" type="xs:string"/>
                    <xs:element name="SourceName" type="xs:string"/>
                    <xs:element name="SuccessorActivityID" type="xs:string"/>
                  </xs:sequence>
                  <xs:attribute name="ID" type="xs:string" use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="namespace" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A1.2 Workflow Instantiation and Status Updates

Once the workflow template has been stored, the workflow can be instantiated as is shown in the following XML description. To do so, the client must pass the workflow ID along with some instantiation info to the Provenance Service. The instantiation info is an XML string formatted according to the instantiation schema. The Provenance Service returns an instance ID to the client that can then be used to update the status of the workflow. To do so, another XML string needs to be passed to the Provenance Service conforming to the status update schema.

```
<?xml version="1.0"?>
<!-- edited with XMLSpy v2007 rel. 3 (http://www.altova.com) by Andrew Branson
(CERN) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="WorkflowInst">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="DataTask" minOccurs="0">
          <xs:complexType>
            <xs:group ref="statusAttr" minOccurs="0"/>
            <xs:attribute name="DescId" type="xs:string" use="required"/>
            <xs:attribute name="InstanceId" type="xs:string" use="required"/>
            <xs:attribute name="DataElement" type="xs:string" use="optional"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="ExecutionTask" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Argument" minOccurs="0" maxOccurs="unbounded">
```

```
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="Order" type="xs:int"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="EnvironmentVariable" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="Name" type="xs:string"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:group ref="statusAttr" minOccurs="0"/>
            <xs:element name="InputLFN" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="DependencyId" type="xs:string" use="required"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="OutputLFN" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="DependencyId" type="xs:string" use="required"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="DescId" type="xs:string" use="required"/>
          <xs:attribute name="InstanceId" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:choice>
    <xs:attribute name="DescId" type="xs:string" use="required"/>
    <xs:attribute name="InstanceId" type="xs:string" use="required"/>
    <xs:attribute name="OutputLocation" type="xs:string" use="optional"/>
    <xs:attribute name="Timestamp" type="xs:string" use="optional"/>
    <xs:attribute name="Version" type="xs:string"/>
    <xs:attribute name="Ns" type="xs:string" default="&quot;&quot;"/>
  </xs:complexType>
 </xs:element>
 <xs:group name="statusAttr">
  <xs:sequence>
    <xs:element name="Status" type="xs:string"/>
    <xs:element name="StatusReason" type="xs:string"/>
    <xs:element name="Timestamp" type="xs:string"/>
    <xs:element name="OutputLog" type="xs:string" minOccurs="0"/>
    <xs:element name="ErrorLog" type="xs:string" minOccurs="0"/>
  </xs:sequence>
 </xs:group>
</xs:schema>
```