XSEDE Architecture
Level 3 Decomposition

12 December 2012

Version 0.9

Felix Bachmann, Ian Foster, Andrew Grimshaw, David Lifka, Morris Riedel, Steven Tuecke
## Document History

<table>
<thead>
<tr>
<th>Relevant Sections</th>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire Document</td>
<td>0.9</td>
<td>12/12/2012</td>
<td>First public release</td>
<td>XSEDE</td>
</tr>
</tbody>
</table>

---
Table of Contents

1 Introduction ........................................................................................................................................... 5
  1.1 The Purpose of this Document .................................................................................................................. 5
  1.2 Web Services vs. Web/Cloud Approaches to Architecture ....................................................................... 6
  1.3 Structure of this Document ....................................................................................................................... 6
  1.4 Document Management and Configuration Control ............................................................................... 7
  1.5 Relationship to Other SADs .................................................................................................................... 7
  1.6 Process for Updating this SAD ................................................................................................................ 7

2 Architecture Background ......................................................................................................................... 8
  2.1 Access layer ........................................................................................................................................... 9
  2.2 Services Layer ..................................................................................................................................... 10
  2.3 Resource Layer ................................................................................................................................... 11

3 Access Layer Level 3 Decomposition ....................................................................................................... 12
  3.1 Thin-client graphical user interfaces ...................................................................................................... 12
    3.1.1 XSEDE User Portal thin-client GUI ............................................................................................... 12
    3.1.2 OAuth Login thin-client GUI ........................................................................................................... 13
    3.1.3 Globus Nexus thin-client GUI .......................................................................................................... 14
    3.1.4 Globus Transfer thin-client GUI ..................................................................................................... 15
    3.1.5 Globus Storage thin-client GUI ....................................................................................................... 16
    3.1.6 Globus Compute thin-client GUI .................................................................................................... 17
    3.1.7 Globus Help thin-client GUI ........................................................................................................... 17
  3.2 Command-line interfaces (CLI) ............................................................................................................... 18
    3.2.1 (GSI)SSH to Globus Online CLI ...................................................................................................... 18
    3.2.2 Globus Toolkit command-line tools ............................................................................................... 20
    3.2.3 Genesis II command-line tools ....................................................................................................... 21
    3.2.4 The UNICORE 6 command-line tools ............................................................................................ 22
  3.3 Thick-client graphical user interfaces .................................................................................................... 24
    3.3.1 UNICORE Rich Client ...................................................................................................................... 24
    3.3.2 Genesis II GUI ............................................................................................................................... 27
    3.3.3 Kepler GUI using Open Standard Web Service Interfaces ............................................................ 32
  3.4 File Systems ........................................................................................................................................... 33
    3.4.1 The Global Federated File System (GFFS) ...................................................................................... 33
    3.4.2 Mounting GFFS ............................................................................................................................. 34
  3.5 Application Programming Interfaces .................................................................................................... 36
    3.5.1 Web Services APIs ........................................................................................................................... 37
    3.5.2 SAGA: A Standard API for Grid Applications ............................................................................... 37
    3.5.3 jGlobus Java API ............................................................................................................................. 39
    3.5.4 Globus Online Java client API ......................................................................................................... 39
    3.5.5 Globus Online Python client API ..................................................................................................... 39
XSEDE Architecture Level 3 v.0.9
5.3.4 File Create ........................................................................................................... 91
5.3.5 File Delete ........................................................................................................... 92
5.3.6 Create Export ..................................................................................................... 93
5.3.7 Write to a replicated file (Tesla) ......................................................................... 94
5.3.8 Write to a replicated directory (Bell) .................................................................. 94
5.3.9 Run a job directly on a BES ............................................................................. 95
5.3.10 Qsub a job on a grid queue (GQ) ..................................................................... 96
5.3.11 Execute a DAGMAN workflow on workflow engine ....................................... 97
5.3.12 Quality Attribute - Availability ....................................................................... 97
5.3.13 Quality Attribute – Performance ................................................................... 99
5.4 The Package View ............................................................................................... 99
5.4.1 UNICORE 6 Server Package ........................................................................... 101
5.4.2 UNICORE 6 client package ............................................................................. 102
5.4.3 Genesis II server container package ................................................................ 102
5.4.4 Genesis II Access Layer (client) Package ......................................................... 104
5.4.5 UNICORE 6 Access Layer Package .................................................................. 105
6 XUAS ...................................................................................................................... 105
6.1 XUAS Services Layer Interfaces ........................................................................ 105
6.1.1 XUAS Execution Management Service Interfaces ......................................... 106
6.1.2 XUAS Data Management Service Interfaces .................................................. 106
6.1.3 XUAS Discovery and Information Service Interfaces ...................................... 109
6.1.4 XUAS Identity Service Interfaces .................................................................... 110
6.1.5 XUAS Accounting and Allocation Service Interfaces ...................................... 110
6.1.6 XUAS Help Desk and Ticketing Service Interfaces .......................................... 110
6.2 XUAS Services Layer Component Types .......................................................... 110
6.2.1 XUAS Execution Management Component Types ......................................... 111
6.2.2 XUAS Data Management Component Types .................................................. 111
6.2.3 XUAS Identity Component Types ................................................................... 112
6.3 XUAS Packages .................................................................................................. 113
6.3.1 Globus Connect package .................................................................................. 113
6.3.2 Globus Connect Multi User package .................................................................. 114
7 XSEDE Architecture Common Elements .............................................................. 114
7.1 Services Layer Interfaces (Common) .................................................................. 114
7.1.1 Data Management Service Interfaces (Common) .......................................... 114
7.2 Services Layer Component Types (Common) ...................................................... 115
7.2.1 Data management components (Common) ...................................................... 115
7.3 Packages (Common) ........................................................................................... 115
7.3.1 Globus GridFTP native packages .................................................................... 115
8 Deployment ............................................................................................................ 116
8.1 Deployed at Service Providers

8.1.1 Genesis and UNICORE packages

8.1.2 Grid Interface Unit

8.1.3 Common components

8.1.4 GRAM5 server

8.2 Deployed as centralized services

8.3 Deployed on clients

References
1 Introduction

The eXtreme Science and Engineering Discovery Environment (XSEDE) provides researchers, educators, and students across the United States and beyond with access to powerful computers, data, tools, and other resources to conduct research and improve the planet.

The XSEDE software architecture defines the mechanisms that people from many different backgrounds can use to access XSEDE services, contribute resources to XSEDE, and develop tools that make use of XSEDE capabilities. An important goal for the XSEDE project is that this architecture be clearly defined so that any member of the XSEDE community can always determine exactly what capabilities XSEDE provides and how those capabilities are to be accessed. The XSEDE requirements engineering activity then allows capabilities to be mapped to requirements so that XSEDE management and the XSEDE user community can determine whether the architecture meets requirements and, as and when required, introduce new requirements that can then be used to drive architectural modifications.

In designing, documenting, and evaluating the XSEDE software architecture, we have sought to follow best practices from the software engineering community [26]. In that community, the term “software architecture” denotes the structure or structures of a software system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [17]. “Externally visible” properties refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, and shared resource usage. We describe the architecture in terms of views, wherein a view provides a representation of a whole system from the perspective of a related set of concerns, or a representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them.

1.1 The Purpose of this Document

This document describes the Level 3 decomposition of the XSEDE architecture. As such, this document complements and greatly extends the Level 1 and 2 decompositions presented in a separate document [13]. As summarized in §2 below, that document defines a high-level decomposition into an access layer, a services layer, and a resources layer (the Level 1 decomposition), and the basic capabilities required at each of those layers (the Level 2 decomposition).

A key concept in the XSEDE architecture is that software components in the access layer interact with software components in the services layer using a well-defined set of interfaces. (Services layer components then interact with resources via various resource-specific interfaces; those details are less important.) The Level 3 decomposition described here further defines:

- An initial set of access layer components;
- The services layer interfaces that access layer components use to access services;
- The services layer components that provide those interfaces; and
- The services layer packages that implement those components in a manner suitable for deployment.

By thus separating access layer from services layer, and interfaces from components and packages, the XSEDE architects aim to achieve clarity in terms of available capabilities; flexibility in that alternative implementations of a specific interface can be substituted; and extensibility in that new resources can be integrated by implementing specified interfaces.
The intended audiences for this document include:

- The XSEDE software development and integration (SD&I) team, who should use it to determine what interfaces components are required to support and what quality attributes they must satisfy;
- The XSEDE operations team, who should use it to determine the deployment requirements associated with various components;
- The XSEDE security team, who should use it to determine whether the XSEDE architecture satisfies security requirements;
- Application developers who want to know what interfaces they should use in order to access XSEDE services; and
- Software developers who want to provide services that are “compliant” with XSEDE specifications. (By compliant we mean that services meet the interface specifications and can interoperate with other such services.)

1.2 Web Services vs. Web/Cloud Approaches to Architecture

The services layer interfaces that access layer components use to access service layer capabilities are arguably the most important element of the XSEDE architecture. Thus, it is not surprising that we encounter hard choices as how to precisely define those interfaces. Two different approaches have been proposed. In the Web Services approach, interfaces are defined in terms of Web Services concepts and technologies that are popular in enterprise architecture. In the Web/Cloud approach, interfaces are defined in terms of concepts and technologies popular in the commodity Web and Cloud. Each approach has strong proponents. Thus, the XSEDE architecture defined here allows for both, as follows:

- The Web Services architectural approach is realized by a set of services layer interfaces, components, and packages collectively known as the XSEDE Wide Area Virtual Environment (X-WAVE): see §4.1 and §5. This architectural subset uses primarily Web Services specifications (e.g., SOAP, WSDL, WS-Security, WS-Trust) in their interfaces and Genesis II and UNICORE components in their implementations.
- The Web/Cloud architectural approach is realized by a set of services layer interfaces, components, and packages collectively known as the XSEDE User Access Services (XUAS, pronounced “Zeus”): see §4.2 and §6. This architectural subset primarily uses Web specifications (e.g., REST, HTTP, SSL/TLS, OAuth) in their interfaces and Globus Online components in their implementations.

Architectural elements that are common to both approaches, such as GridFTP and MyProxy, are described in §7.

1.3 Structure of this Document

This document is organized as follows:

1. Introductory material.
2. Background material on the Level 1 and 2 decompositions defined elsewhere.
3. An initial set of XSEDE Access Layer components of different types.
4. An introduction to the Web Services and Web/Cloud approaches to architecture.
5. X-WAVE interfaces, services, and components
6. XUAS interfaces, services, and components
7. Common interfaces, services, and components
8. Some initial material on deployment.

1.4 Document Management and Configuration Control
This Version 0.9 of the XSEDE Level 3 Decomposition was released on October 11, 2012.

1.5 Relationship to Other SADs
This document builds on the Level 1 and 2 Decomposition presented in another document [13].
Various documents will eventually indicate how the elements described in the XSEDE architecture are used to satisfy requirements arising in different contents.

1.6 Process for Updating this SAD
Please email architects@xsede.org with any comments and suggestions on any aspect of this document, its contents, and the presentation. The XSEDE Architecture Team will respond promptly and acknowledge any input that results in changes to future versions of this document.
2 Architecture Background

The Level 1 decomposition of the XSEDE architecture [13] defines three “layers”: an access layer, a services layer, and a resources layer. See Figure 1. Software components in the access layer interact with software components in the services layer using a well-defined set of interfaces. These interfaces provide access layer components with abstractions that mask the heterogeneity of the underlying resources layer. How the services layer components are implemented is immaterial. Typically though, the services layer components will interact with resources via a variety of resource-specific interfaces.

Figure 1. The three layers of the XSEDE architecture. (From [13].)
In the remainder of this section we summarize the Level 2 decomposition of the XSEDE architecture, providing brief descriptions of the primary functions that are required in each layer. In later sections, we provide the details of the XSEDE architecture's Level 3 decomposition, which include the various interfaces used, the components that implement those interfaces, and the packages of components that are used in deployments.

2.1 Access layer

Recall from the Level 1 and 2 description of the XSEDE architecture that the access layer consists of a number of different types of access mechanisms targeted at different usage modalities.

Access layer components fall into five broad categories: thin-client graphical user interfaces (GUIs), thick-client GUIs, command-line interfaces (CLIs), application programming interfaces (APIs), and file system mechanisms (Figure 2).

Thin-client graphical user interfaces (GUIs) are access layer components that are accessed via a web browser and thus do not require users to install any software beyond a standard Web browser on their workstation/desktop/PDA in order to access XSEDE resources.

Thick-client graphical user interfaces (GUIs) are an alternative to thin clients. These GUIs require that some application beyond a Web browser be installed and executed on the machine from which XSEDE services are to be accessed. For example, a thick client may be installed on an XSEDE Service Provider (SP) login node, on a departmental file server, on the user's desktop, or on the user's PDA.

Command-line interfaces (CLIs) are tools that allow XSEDE resources and services to be accessed from the command line or via scripting languages such as BASH.

Figure 2. The access layer consists of a diverse set of components that facilitate access to XSEDE services. The items listed in “notes” are examples in each category. (From [13].)
Application programming interfaces (APIs) provide language-specific interfaces for interacting with XSEDE services. APIs are implemented by libraries that can be linked with application programs.

File system mechanisms are interfaces to XSEDE services and resources that are provided through the file system paradigm and POSIX (Portable Operating System Interface) interfaces.

### 2.2 Services Layer

The services layer implements the interfaces that both users and other services invoke, typically via access layer components. Some services are deployed at many or all XSEDE SPs (e.g., file access services), while others may be deployed in just one location (e.g., the XSEDE User Portal). The XSEDE Architecture Level 3 Decomposition described in later sections of this document details XSEDE services and their capabilities (Figure 3).

![ cmp Components](cmp Components)

**Figure 3. The services in the services layer are decomposed into seven broad categories: Execution Management, Discovery and Information, Identity, Accounting and Allocation, Data Management, Infrastructure Services and Mechanisms, and Help Desk and Ticketing. (From [13].)**

**Execution Management Services** are concerned with instantiating, and managing to completion, units of work that may consist of single activities, sets of independent activities, or workflows.

**Discovery and Information Services**: Discovery and information services address needs to find resources based on descriptive metadata and to subscribe to events or changes in resource status.

**Identity**: Identity services provide the means to identify and provide attributes about individuals, services, groups, roles, communities, and resources (e.g., a particular software component or set of computers). This information is often used for making authorization decisions, maintaining audit trails, logging, accounting, and searching for resources owned by, or accessible to, a particular individual, service, or group.

**Accounting and Allocation**: Accounting and allocation is concerned with keeping track of resource consumption and what consumption is allowed.
Data Management: Data management includes the access and manipulation of both user data and system data residing at service providers, including XSEDE resource providers, campuses, research labs, other infrastructures, and third-party providers such as cloud storage providers.

Infrastructure Services: Infrastructure services include naming and binding services, resource introspection and reflection services, and fault detection and recovery services.

Help Desk and Ticketing: Help desk and ticketing services provide interfaces for ticket management (e.g., submit, check status, change status, comment, query) and for help desk federation (e.g., ticket forwarding and tracking).

2.3 Resource Layer

A resource is a physical or virtual resource to which we wish to enable remote access via a services layer protocol (Figure 4). Examples of resources include compute servers (with associated queuing systems), file systems, relational databases, scientific instruments, a trouble ticket database, and wide area networks. Two other, perhaps less obvious examples of resources are a virtual organization and an advance reservation; like the other examples, both have some state that XSEDE users may wish to access and manipulate.

Resources are often accessed and manipulated by local, non-standardized methods. For example, in the case of compute servers, we have schedulers such as LSF, Torque, and SGE. As discussed above, the services layer both enables remote access to, and hides heterogeneities associated with, those idiosyncratic access methods.

Figure 4. Examples of resources include archives, file systems, job queues, and other software and hardware systems that are virtualized by the services layer. (From [13].)

For example, the SGE “normal” queue on Ranger at the Texas Advanced Computing Center is a job queue resource. Similarly the archive (or a part of it) at the Pittsburgh Supercomputer Center is an archive resource, the Lustre file system at Indiana University is a file system resource, and the XSEDE Kerberos realm is an identity resource.
3  Access Layer Level 3 Decomposition

XSEDE serves a diverse and heterogeneous user community. The researchers, educators, and students who use XSEDE services and resources not only belong to various scientific disciplines but also possess varying levels of experience with tools such as MPI [42], queuing systems, scripting languages, X.509 certificate systems, data movers, and debuggers. Therefore, XSEDE is not a “one size fits all” system but rather addresses the needs of several distinct types of users. This approach is especially apparent in the access layer, which we have designed to allow different types of users to interact with XSEDE in ways similar to those with which they are already familiar.

Accordingly, rather than apportioning the user communities by discipline, we divide them by role and corresponding level of high-performance computing (HPC) sophistication: 1) novice users, who are used to interacting with complex systems via canned applications, gateways, GUIs and the file system interface on their desktop; 2) application developers (e.g., gateway developers) who prefer APIs and have a much better understanding of the underlying system model; 3) systems administrators who manage, configure, and perform fault analysis; and 4) expert computational scientists who live on the bleeding edge, who are systems savvy, and strive to optimize performance.

For expert computational scientists, the current ssh and Unix-based environments may often be adequate. Simple changes to that environment proposed by XSEDE, such as unifying the shell environment (e.g., shell variable names, common tools) and providing high-performance and reliable data movement services allow these users to be more productive, but generally speaking, the key to serving this class of users is to keep out of the way.

For other users, XSEDE’s goal is to adapt to the user’s needs, rather than expecting the user to adapt to XSEDE; as most scientists do not want to become computer hackers, XSEDE must support interaction modalities and paradigms with which users are already familiar. Toward that end, simplicity of access is a major architectural goal and motivates our emphasis on the access layer.

In this section, we describe the various access layer components that are included in the XSEDE architecture at this time. For each, we provide a brief description, characterize its target users, give examples of its use, provide example screen shots (where appropriate), list other components on which it depends, and provide a brief deployment view.

3.1  Thin-client graphical user interfaces

As noted in §2.1, thin-client graphical user interfaces (GUIs) are access layer components that are accessed via a web browser and thus do not require users to install any software beyond a standard Web browser on their workstation/desktop/PDA in order to access XSEDE resources.

We describe here seven such thin-client GUIs: the XSEDE User Portal, OAuth Login, Globus Nexus, Globus Transfer, Globus Storage, Globus Compute, and Globus Help.

3.1.1  XSEDE User Portal thin-client GUI

Description: The XSEDE User Portal (portal.xsede.org) provides XSEDE users with access to view and manage their accounts and allocations, as well as find information about and access XSEDE services and resources.

Target user: Novice users or expert computational scientists who want information on XSEDE system status, etc.

Example screenshots: Figure 5 shows the XSEDE User Portal thin-client GUI.
3.1.2 OAuth Login thin-client GUI

**Description:** OAuth 2.0-compliant, Web browser-based authentication to a security domain, which returns an access token granting access to a set of resources.

**Target user:** A user who wants to connect to a website that uses the OAuth protocol for federated identity login (e.g., with the XSEDE identity provider) such as Globus Online services (e.g., Globus Nexus, Globus Transfer) or a third-party Web site that uses CILogon. This user may be a novice or an expert; both are likely to use Globus Transfer, for example, for data movement.

**Uses:**
- OAuth 2.0 interface (§6.1.4.1).

**Example screenshot:** See Figure 6.
Figure 6. XSEDE OAuth Login thin-client GUI.

Deployment view:

- One instance (with redundancy for availability) installed in front of the XSEDE Kerberos domain.
- One instance (with redundancy for availability) as part of CILogon (for InCommon identities).
- One instance (with redundancy for availability) in front of any other identity domain required to reach resources (e.g., non-InCommon campus security domains). See the Globus Connect Multi User package (§6.3.2) as one way to deploy this.

3.1.3 Globus Nexus thin-client GUI

Description: The Globus Nexus Web GUI provides three main functions:

- **Federated identity management**: Manages binding of external federated identities with Globus/XSEDE identities
- **Group management**: Allows users to create groups and manage group membership and policies
- **Profile management**: Allows users to manage profile attributes associated with their Globus/XSEDE identities

The Globus Nexus GUI provides for customization with different skins for different user communities; a featured used, for example, by Blue Waters and BIRN.

Target user: A user who wants to: log in to a Globus Online service (e.g., Globus Transfer, Storage) or a third-party website that uses Globus Nexus; self-manage bindings of federated identities to a Globus Nexus identity; self-manage group creation, configuration and membership; and self-manage profile attributes associated with their Globus account. This user may be a novice or an
expert; both are likely to use Globus Transfer for data movement and/or Globus Nexus for group management.

*Example screenshots:* The screenshot in Figure 7 illustrates Globus Nexus thin-client GUI support for federated identity management. The user in question has registered three credentials with Globus Nexus: a Google OpenID credential, an SSH credential, and an XSEDE credential.

![Globus Nexus Thin-Client GUI](image)

*Figure 7. Example Globus Nexus GUI screenshot. (See text for details.)*

*Uses:*

- XUAS Nexus REST interface (§6.1.4.4).
- OAuth Login thin-client GUI (§3.1.2).

*Deployment view:* Provided by a single, global Globus Nexus component.

### 3.1.4 Globus Transfer thin-client GUI

**Description:** File and folder transfer, synchronization, and management.

*Target user:* A user who wants to transfer, synchronize, and manage files and folders on XSEDE service providers’ and other connected storage resources, including campus servers, scientific instruments cloud providers, and personal computers and laptops. See the descriptions of the GridFTP (§7.3.1), Globus Connect (§6.3.1), and Globus Connect Multi User (§6.3.2) packages for information on how to connect a storage resource to Globus Transfer. This user may be a novice or an expert; both are likely to use Globus Transfer for data movement.

*Example screenshot:* The screenshot in Figure 8 illustrates Globus Transfer thin-client GUI support for transferring files. Note the symbolic endpoint names (xsede#trestles and ci#pads), the list of directories and files at each endpoint, and the transfer options at the bottom. (Checksum-based synchronization and encrypted transfer are selected.) Other screens allow users to view and manage transfer activities and manage endpoints.
Figure 8. Example Globus Transfer thin-client GUI screenshot. (See text for details.)

Uses:

- Globus Nexus thin-client GUI (§3.1.3).
- XUAS Transfer REST interface (§6.1.2.2).

Deployment view: Provided by a single, global Globus Transfer component.

3.1.5 Globus Storage thin-client GUI

Description: Globus Storage provides novice users with cloud storage abstractions similar to Dropbox, but with features specifically suited for big-data, collaborative science. In particular: it allows for data to be stored and replicated on one or more object stores that may be located at research facilities, campuses, or for-fee cloud object stores (e.g., SDSC data cloud, Amazon S3); it supports Dropbox-like synchronization and http interfaces, as well as Globus Transfer, which is already widely adopted by DOE and NSF research infrastructures, for high-performance, secure file transfer and synchronization between its object stores and external systems such as supercomputers, scientific instruments, analysis clusters, etc.; it integrates with Globus Nexus identity management, allowing single sign-on support from widely adopted federated identity systems such as InCommon; it supports user-controlled sharing of data with users and groups defined in Globus Nexus; it automatically retains old versions of files that are modified, subject to configurable, time-based retention policies; and it supports easy creation of read-only snapshots of data sets for publication to larger communities.
Target user: A user who wants cloud storage that is well integrated with XSEDE service providers. This user may be a novice or an expert; both are likely to use Globus Storage.

Uses:
- Globus Nexus thin-client GUI (§3.1.3).
- XUAS Storage REST interface (§6.1.2.3).

Deployment view: Provided by a single, global Globus Storage component

3.1.6 Globus Compute thin-client GUI

Description: High-throughput compute management.

Target user: A user who wants a simple Web interface to which they can hand off the task of defining, submitting, managing, and monitoring individual tasks and many-task (high-throughput) computing workflows that may execute on XSEDE, campus, cloud, or personal compute resources. This user may be a novice or an expert; both are likely to use Globus Compute for compute movement, although perhaps for somewhat different purposes. The novice may use it to submit a single job, but the expert to manage many jobs.

Availability: This component is not yet available.

Uses:
- Globus Nexus thin-client Web GUI (§3.1.3).
- XUAS Compute REST interface (§6.1.1.2).

Deployment view: Provided by a single, global Globus Compute component.

3.1.7 Globus Help thin-client GUI

Description: Web GUI help desk for all Globus Online services. (Note: The service also provides an email interface, mailto: support@globusonline.org.)

Target user: A user who needs help with Globus Nexus, Transfer, Storage, Compute, etc. This user may be a novice, systems administrator, or expert computational scientist.

Example screenshot: Figure 9 shows some of the options available from the Globus Help GUI.
3.2 Command-line interfaces (CLI)

As noted in §2.1, CLIs are tools that allow XSEDE resources and services to be accessed from the command line (e.g., in the Windows command shell) or via scripting languages such as BASH.

We describe three sets of CLIs: the (GSI)SSH CLI functions; Globus Toolkit CLIs (globus-url-copy, globus-run, and related commands); the Genesis II CLI; and the UNICORE 6 CLI.

3.2.1 (GSI)SSH to Globus Online CLI

Description: All Globus Online CLI functionality is provided via a restricted shell operated by Globus Online, to which users can issue commands via ssh and gsissh. In this way, Globus Online avoids the need for installation of client-side software (beyond the quasi-ubiquitous ssh). See the XUAS service layer interfaces (§6.1.2.4, etc.) for descriptions of the CLI commands that can be accessed via this interface.
**Target user:** The Globus Online CLI is recommended for users who wish to write scripts to interact with remote resources, as when performing a series of actions that must frequently be repeated, for example. For one-off interactions, Globus Online thin-client GUIs (§3.1) may be preferred.

**Examples:** A Globus Online CLI call has the following general form for ssh and gsish:

```plaintext
ssh <user>@cli.globusonline.org <command>
gsish <user>@cli.globusonline.org <command>
```

where `<user>` denotes the user’s Globus Online username and `<command>` is a Globus Online command. See §4.2.1 for detailed examples.

**Example screenshots:** Figure 10 illustrates interactive use of the Globus Online CLI. The user first SSHs to the CLI server and then issues two commands: help, to list available commands, and status, to list brief status information on currently active tasks (of which there is one).

Note that the CLI also can be used in scripting mode, as illustrated in the following example. Here, `$` is the command prompt. In the first line, the user issues a status command (using the “-O kv” option to specify that its output should be in key-value format) and pipes its result to grep to extract output lines that pertain to ACTIVE tasks. The result, on the second line, includes taskid, request time, type of command used to start the task, user-supplied label, and status.

```plaintext
$ ssh tuecke@cli.globusonline.org status -O kv | grep ACTIVE
taskid=fcdcaffe-b0f2-llel-bf56-1231380b8963 request_time=2012-06-07+22:49:43Z command=API+0.10+GO label=More+Software status=ACTIVE
$
Figure 10. SSH to Globus Online CLI for interactive use

Uses:
- XUAS command-line interface (§6.1.2.4).

Deployment view: Provided by a single, global XUAS CLI component, accessible via (gsi)ssh client on the user's computer.

3.2.2 Globus Toolkit command-line tools

The Globus Toolkit provides file transfer and job management CLI programs.

3.2.2.1 globus-url-copy file transfer CLI

Description: This scriptable command-line tool can perform multi-protocol data movement between two locations, either or both of which can be local or remote. It supports gsiftp:// (GridFTP), ftp://, http://, https://, and file:/// protocol specifiers in the URL. For GridFTP, globus-url-copy supports all implemented functionality. Versions from GT 3.2 and later support file globbing and directory moves.

Target user: globus-url-copy is intended for use by application developers, systems administrators, and expert computational scientists. However, the Globus Transfer CLI (or Web GUI) is recommended in preference to globus-url-copy.

Specification: See the GT 5.0.0 GridFTP User’s Guide [5].

Deployment view: This program must be installed on the user's computer. Depending on the site, the software may be installed by a systems administrator in a standard location or installed by the user in their local file system space.
3.2.2.2  globusrun and globus-job-* job management CLI

Description: These scriptable command-line tools can perform remote job submission and management to a Globus Toolkit GRAM5 service.

Target user: These commands are intended for use by application developers, systems administrators, and expert computational scientists.

Specification: See the GT 5.0.0 GRAM5 User’s Guide [4].

- globus-job-run – Execute a job using GRAM, and wait for completion
- globus-job-submit – Submit a batch job to a GRAM5 service
- globus-job-status – Check the status of a GRAM5 job
- globus-job-cancel – Cancel a GRAM batch job
- globus-job-clean – Cancel and clean up a GRAM batch job
- globus-job-get-output – Retrieve the output and error streams from a GRAM job
- globusrun – The program behind the globus-job-* commands, which also can be used directly.

Deployment view: These programs must be installed on the user’s computer. Depending on the site, the software may be installed by a systems administrator in a standard location or installed by the user in their local file system space.

3.2.3  Genesis II command-line tools

The UNICORE 6 and Genesis II command-line tools overlap significantly in functionality at this moment. Both use as their model the command shell from Unix, with ls, cat, cp and run commands (e.g., grid ls or ucc ls). The Genesis II team plans to work with the UNICORE 6 team to merge the command-line interfaces into a single, consistent tool. This tool will make available the common command-line commands, including file and directory commands, security and identity commands, job and queue management commands, and other miscellaneous commands.

The command-line tools in Genesis II are shown below. New command-line tools can be added by creating a Java jar file that implements the grid_runable interface and placing the jar file in the appropriate directory. Short descriptions of each command are available via "help <command>" in the shell and complete descriptions via "man <command>". Complete documentation is available at: http://genesis2.virginia.edu/wiki/uploads/Main/GenesisII_omnibus_reference_manual.pdf.

3.2.3.1  File and Directory Commands

- cat - Concatenates, or displays, the data from one or more grid or local resources.
- cd - Changes the current working RNS path of the current grid shell or session.
- cp - Copies data between local file-system files and/or grid RNS namespace ByteIO resources.
- echo - Echoes the contents of the command line.
- export - Exports an entire local directory structure with files into the grid giving users access to that data through the grid without requiring copies of the data to be made.
- ln - Creates new entries in RNS space for existing resources either by linking them from other RNS paths, or by creating them from EPRs.
- ls - Lists entries in a directory.
- mkdir - Creates a directory.
- pwd - Returns the current path of the user.
- rm - Deletes/Removes entries.
• unlink - Unlinks — or removes the namespace entry for — an entry in the current session’s RNS space. Can also be used to remove local entries.

3.2.3.2 Security Commands

• authz - Inspects and modifies the authorization policy for a grid resource.
• chmod - Modifies the authorization policy for a GenesisII grid resource.
• create-user - Creates a new grid user from scratch.
• create-user-delegate - Creates a new grid user by delegating an existing credential.
• Idp - Creates an identity-provider resource (IDP) from which users can obtain credentials.
• login – Log in (or acquire credentials) that can be used in the grid.
• logout - Removes authentication information from the user's context.
• whoami - Prints out the credentials of the currently logged in user.
• certTool – Is a separate command-line tool that allows users to create or import certificates and/or keypairs within keystore files.
• xsedeLogin – Authenticates against the XSEDE Kerberos realm and acquires a myproxy certificate for the client session.

3.2.3.3 Application and Execution Commands

• qconfigure - Configures the number of slots available for resources managed by grid queues.
• qkill - Terminates (moves to a final state) a job in a grid queue.
• qstat - Lists the public information about all jobs currently managed by a grid queue.
• qlist - Lists the public and private information about jobs currently managed by a grid queue and owned by the caller.
• qsub - Submits a new job to a grid queue.
• run - Runs a job on the grid.
• bes-policy - Allows you to query or manipulate the BES policy currently being enforced on a BES container.
• tty - Allows you to watch or unwatch an existing TTY object.

3.2.3.4 Miscellaneous Commands

• connect - Connects the current session to an existing net.
• create-resource - Creates a new resource using generic creation mechanisms.
• ftpd - Creates an FTP Daemon on the client machine that can be used to access FTP resources with an FTP client.
• get-attributes - Retrieves and prints the attribute document for a target.
• get-bes-attributes - Retrieves and prints the attribute document for a BES container.
• ping - Sends a character string to a resource, which is echoed back.
• schedule-termination - Schedules a resource to auto-terminate at some time in the future.
• script - Executes a Genesis II XML script.

3.2.4 The UNICORE 6 command-line tools

The UNICORE Command-line Client (UCC) offers a set of basic commands to work with UNICORE 6. The basic functionalities include running a job, getting their output, and transferring files to the UNICORE 6 servers.

Apart from this basic functionality, it offers a wide variety of extension mechanisms with hooks that enable an easy integration of other WS client interfaces (e.g., the UNICORE Common Information Service client). Also, it offers job description methods using JSON, a “batch processing mode” for
running many jobs and scripting support using Groovy. Support for the UNICORE6 workflow system is also available. A five-minute tutorial is available.

Figure 11. UNICORE 6 Command-line Client and some of its options
3.3 Thick-client graphical user interfaces

As noted in §2.1, thick-client GUIs require that some application beyond a Web browser be installed and executed on a machine from which XSEDE services are to be accessed. We describe two such thick-client GUIs here, the UNICORE Rich Client, and the Genesis II GUI.

3.3.1 UNICORE Rich Client

The UNICORE Rich Client (URC) includes a GUI interface for the user that provides a graphical workflow editor, grid browser, workflow execution monitor, service details window, and a user-definable, application-specific interface. Application-specific enhancements are provided with JAVA GridBeans. Job definition within the URC is compliant with the JSDL standard [12]. The various sections below describe each of the key functionalities in more detail with examples.

Figure 12 illustrates the overall view of the UNICORE Rich Client. It is well embedded within Eclipse and as such it can be combined with other existing Eclipse-based solutions (e.g., Parallel Tools Platform from IBM) or changed in look and feel according to the needs of users.

The general layout provides a Grid Browser with an overview of all resources and services on the left side. Other views include the TrustStore, to easily configure security credentials. The Log monitor can be used for detailed views on execution and state changes. In contrast, the right side of the URC often is used to create and configure a single computational job or create a workflow. All jobs and workflows can be saved for reusability and sharing across a scientific group.

Figure 12. UNICORE Rich Client Interface.
3.3.1.1 Job Execution and Workflow Management

The Rich Client can be used for the execution of simple jobs to one HTC or HPC resource. The execution of workflow nodes from the editor as shown in Figure 13 can be done using the standard specification BES [39] as well. The GUI enables different views from Beginner to Expert users to lower the barrier for end users who have not been previously engaged using X-WAVE.

Figure 13. UNICORE 6 Rich Client Workflow Tool. The workflow tool is an Eclipse plug-in.

Another important aspect related to execution is the support of existing installed applications at XSEDE resources: see, for example, the bottom-left corner of Figure 13. Examples are Autodock or CMOPAC, applications that have been previously configured on the service provider and are available for an easy execution.

The installed applications appear in the job creation view on the right side of the URC and easily can be used via drag and drop.

3.3.1.2 GridBean: Extensibility for Scientific and Engineering Applications

Scientific or engineering-specific application GUIs can be embedded easily in the URC using a plug-in concept named GridBeans. A generic GridBean enables easy access to command-line-oriented application packages installed on XSEDE resources.

The GridBeans also are able to take advantage of the scientific or engineering scientific packages that are available at different XSEDE resources. Figure 14 and Figure 15 illustrate examples from various scientific fields. More details on the extensibility of the UNICORE Rich Client are provided elsewhere in this document [30].
3.3.1.3 URC Terminal Access

Another feature used is the terminal access of the URC, as illustrated in Figure 16. This view enables the access via SSH on resources in a single sign-on manner. It is typically used for checking intermediate results or to validate data related to computational activities.

Different SSH types are supported, reaching from the GSISSH up to plain SSH connections. Once URC is started, users need to enter a password to unlock their credentials. After that, no further password is required to enter via SSH any configured XSEDE system.
3.3.2 Genesis II GUI

The Genesis II GUI is a component of the Genesis II client tool described earlier. The GUI has many capabilities implemented via subtools. These include (but are not limited to):

- A browser for the XSEDE GFFS directory namespace.
- A command shell that supports history, command completion, etc.
- Tools to define, execute, and manage sequential, parameter space, and parallel MPI jobs on individual compute resources (e.g., Ranger) and on grid queues that aggregate compute resources and perform resource matching and scheduling of jobs on those compute resources.
- Tools to create and manage identities and groups.
- Tools to share data, such as directories trees on SP file systems, campus file systems, and lab file systems, with others.
- Tools to examine and configure access control policies.

Below we briefly look at four of these subtools — the browser, grid queue manager, access control list manager, and job submission definition tool.

3.3.2.1 The Browser

The browser allows the user to browse the GFFS directory structure, set access controls, drag and drop files between the GFFS and their desktop, and access the other tools.

Figure 16. Terminal access of the URC to XSEDE resources.
Figure 17. The browser presents the user with a familiar interface to interact with the resources and services in XSEDE. Here we show the tearing off of a new browser window.

3.3.2.2 The Grid Queue Manager
The grid queue manager tool within the Genesis II GUI displays all of the users’ jobs, their state, execution location (if they have been scheduled), and a link to the error log (if they have failed). The tool also can be used to add resources to the queue, set job limits on those resources, and examine resource states.
Figure 18. Grid Queue manager interface showing jobs in the queue. The job ticket, user-provided job name, submit time, submitter, number of attempts to run the job so far, and job status are shown. The lower window shows the resources on which the queue will schedule jobs and how many slots may be used.

3.3.2.3 Job History Tool

Figuring out what happened when a job fails can be a challenge when jobs are running on many different resources in many different places. The job history tool is invoked by right clicking on a job in the queue manager interface. The user can select a level of detail. For example at the highest level of detail the user can see when each file staging activity began, how many bytes were transferred, the exit code returned by the application, and so on.
Figure 19. The job history tool displays details on a job's lifetime at several levels of detail. The finest level of detail is shown here.

3.3.2.4 Access Control List Manager Tool

Other available tools include the ACL (Access Control List) Manager Tool, which allows resource owners to modify the access control lists of owned resources such as compute resources, files, directories, queues, and groups. Principals being given access do not necessarily have to have an account on an SP resource. A file owner, for example, may choose to give a colleague direct access to a particular file based on a colleague's X.509 certificate, or based on a username/password they create on the spot.
Figure 20. Access Control List Management GUI. The uvaIDPgroup, mmm2a, and jfk3w (among others) have read access to this file.

### 3.3.2.5 JSDL (Job Submission Description Language) Tool

JSDL is a standard XML-based language for job definition. As an XML language, JSDL is not fit for (easy) human consumption. The JSDL tool is form-based. Users fill in the forms to specify: the program to execute, the application resource requirements, whether it is sequential or MPI, and the files to stage in and stage out. If the user wants to run the same program with different parameters, he or she can set up iteration spaces for parameter space jobs as well. The tool generates a JSDL file that can then be sent to an execution service.
3.3.3 Kepler GUI using Open Standard Web Service Interfaces

One way in which scientists may use XSEDE resources is via the Kepler workflow engine and its GUI client [10]. The Serpens suite [23] is a set of actors, composites, and workflows, developed by the Poznan Supercomputing and Networking Center (PSNC). This suite provides support for various middleware systems and open standard implementations (e.g., OGSA-BES and JSDL). The suite covers the standard activities related to remote job management, job submission, data handling, identity setups, and several others.

One example showing part of the fusion science community that takes advantage of UNICORE BES interfaces is illustrated in Figure 22. As open standards are used, the provided actors are not limited to UNICORE and can be used with any other adoption of the OGSA-BES standard.
3.4 File Systems

As noted in §2.1, file system access layer interfaces enable access to XSEDE services and resources through the file system paradigm and POSIX (Portable Operating System Interface) interfaces. We describe one such interface here, the Global Federated File System (GPFS). Material on the XSEDE Wide Area File System will be added at a later date.

3.4.1 The Global Federated File System (GFFS)

Simplicity of access is particularly important for novice and new-to-XSEDE users. For novice users, for whom the computer is a tool mainly used for reading email, saving attachments, opening documents, cruising through the directory/folder structure looking for a file, etc., the file system is a familiar friend. The file system paradigm of data access is one of the most ubiquitous in human-computer interaction. Whether you are browsing through folders and double-clicking on files, or using the cd or cat commands to browse through directories and files, the notion of a hierarchical arrangement of files and directories has become second nature to virtually every computer user in the world.

XSEDE implements a data access paradigm that simply extends this familiar process across organizational boundaries and to a wider variety of file types. Specifically, data of all types in many different organizations are mapped into shared name spaces (directory structures) and then mapped into a scientist’s local operating system as a mounted file system. Scientists can then access data (subject to access control) in ways with which they are familiar. Not only can they access it by double-clicking or executing the cat command, their applications also can access the data as if it were local. The same logic of extending a familiar data access paradigm also applies to systems administrators, most of whom in the supercomputer world are familiar with managing Unix/Linux systems via scripts that interact with the file system to parse logs, write configuration files, and manipulate devices via the /dev file system.
As called for in the RFP (NSF 08-571), XSEDE resources are visible via “a single logical namespace.” The namespace is implemented using RNS (Resource Namespace Service [67]), a simple directory specification that maps path strings, e.g., "/GENEBANK/databases/bio/sequences/pir21.sqf” to WS-Addressing EndPoint References (EPRs). An EPR is essentially a Web Services pointer and plays the role similar to an inode in the directory system. The result is a directed graph in which RNS services are the interior nodes and the leaves are other types of services. As one might suppose, the RNS namespace is modeled on file system directory services. To access a resource anywhere in XSEDE, all that is needed is the pathname — XSEDE does the rest.

Many different types of resources can be pointed to by EPRs and thus by pathnames, including files and directories. Less obvious examples include relational databases, execution services, logging services, and secure token services (identities and groups). For example, an execution service that implements OGSA-BES [39] also can implement the RNS directory interface. To view jobs executing on a resource, a simple ls is sufficient. The jobs can appear in the directory as directories themselves, within which the status of the job can be viewed as a file and the current working directory of the job as another directory.

RNS services may be implemented by many different software stacks to model many different types of list or directory services. Naturally these services may “point” to one another, forming a larger graph. The result is that the RNS namespace becomes the architectural mechanism for integrating different software stacks in a way with which people are familiar.

The global namespace provides the mechanism to name any XSEDE resource with a human-readable pathname. Global namespace pathnames are location-, migration-, implementation-, replication- and possibly failure-transparent names that can be used by applications and programmers to name anything. They can be used to specify the queue or meta-scheduler that should be used, which account should be charged, which virtual organization to join, or anything else. The global namespace forms the backbone of the XSEDE global file system. The directory structure of the global namespace is the directory structure of the XSEDE global file system.

### 3.4.2 Mounting GFFS

XSEDE exploits file system familiarity by providing file system drivers for each of its supported platforms. In Linux and MacOS, we provide a FUSE (File System in Userspace) [79] file system driver, and in Windows, an Installable File System (IFS) driver\(^1\). These drivers can be executed on any Linux, MacOS (although FUSE is not available for Lion), or Windows client that has access to the internet. Thus they can be run directly from user desktops, from login nodes on clusters on campuses, and often even from compute nodes on campus and on the SP’s operating system. Figure 23 shows a user application, perhaps simply a bash shell, interacting with the operating system file system and opening, reading, and writing files and directories. These calls are redirected to the GFFS-aware FUSE file system driver. The FUSE driver examines the local cache if appropriate, and then makes the appropriate calls on RNS and ByteIO [59] resources linked into the XSEDE GFFS wherever they are located.

---

\(^1\) The IFS driver needs work to become a production-ready component. A task to harden the driver is in the SD&I task set but has not been scheduled for development.
Figure 23. The GFFS-aware FUSE driver maps a globally visible directory structure consisting of files, directories, databases, instruments, and other file and directory like services into the local file system where they can be manipulated as if they were local. User applications and shell scripts do not need to be modified to use the GFFS; they can run out of the box using resources throughout XSEDE.

The directory structure is arbitrary, but, as in Unix, certain conventions are followed. The current XSEDE directory structure has entries for compute resources, queues, groups, identities, and user home directories.

XSEDE grid resources often implement one or both of the RNS or ByteIO interfaces described later. These interfaces, modeled on traditional POSIX-like APIs, describe respectively a directory-like interface for accessing name-endpoint mappings and a file-like interface for reading and writing data, thus permitting XSEDE to treat each resource as if it were a directory or a file. This makes it relatively easy to implement applications that interact with these resources. Once mounted, access to XSEDE is no different than access to any other file or directory on your local machine.

This access extends beyond simple file data. As many resources in XSEDE implement a file or directory interface, even resource types that do not represent pure data or directory concepts are mapped as files and directories. This notion of treating resources as files and directories first appeared in Plan 9 [68] and today is in use in the /proc file system. Treating grid resources not usually thought of as files or directories as such extends the file system-based access layer into the compute realm, allowing users to manage compute resources (such as jobs, hosts, and queues) using familiar file system abstractions.

For example, resources such as BES containers or grid queues (described later) can also implement the RNS interface. If we list the contents of /compute/UVA/CS/pbs-centurion, we will see the set of jobs currently queued or running on the PBS queue controlling the centurion cluster in the CS department at UVA. If we ls one of those jobs, we will be able to see entries that include a file containing the status and the log for the job, the session directory for the job, and other details about the job.
Similarly, a user can find out what jobs he or she currently has running by listing the contents of the grid queue directory `/queues/grid-queue/jobs/mine/running`.

![Terminal output showing directory listing](image)

**Figure 24. Directory listing of a job running on the centurion cluster at Virginia. Note that most of the directories in the path do not exist on any real file system.**

The directory paradigm maps to other less obvious examples. For example, XSEDE WS-Trust [29, 66] identity resources — called *secure token services* (STS) in the WS-Trust standard — implement both the STS interface as well as the RNS interface. Links can be added from an identity resource to other identity resources, such as group identities. Figure 25 shows the listing of the *morgan* identity, indicating that there are links to the *UVA-idp* group and the *management-group*. Whenever the *morgan* identity is acquired, XSEDE automatically attempts to acquire the *UVA-idp-group* and *management-group* membership identity tokens. Owners of groups can control access to their group or virtual organization simply by changing the access control list of the group resource.

![Terminal output showing morgan identity listing](image)

**Figure 25. Morgan identity entries for group membership.**

### 3.5 Application Programming Interfaces

As noted in §2.1, application programming interfaces (APIs) provide language-specific interfaces for interacting with XSEDE services. APIs are implemented by libraries that can be linked with application programs.
We describe five APIs here: Web Services APIs, SAGA, jGlobus, and Java and Python Globus Online APIs.

### 3.5.1 Web Services APIs

Specification-focused APIs directly access services layer functionality in Java, C, Python, or any other language. XSEDE comes with Java-based Web Services APIs that directly manipulate resources that implement the specifications described in the services layer, e.g., execution services, directory and file services, etc. The Java run-time libraries that provide the APIs also provide utility classes to create and parse the various XML data structures needed for these APIs. Other language APIs, e.g., C/C++, can be generated using any one of a number of tools such as gSOAP, Apache/Axis, and Microsoft wsdl.exe.

Specifically, the XSEDE Java RTL (runtime library) has classes for RNS directories, ByteIO files and streams, OGSA-BES factories, as well as “base class” WS-RF [36] facilities such as notification, lifetimes, and `getresourceproperties`. That said, we do not expect many programmers to use these low-level interfaces, as they require an intimate knowledge of the specifications and they are more likely to change over time than higher-level specifications. Instead, we expect most programmers to access XSEDE resources through high-level APIs (such as SAGA), via the file system, and via command-line tools.

### 3.5.2 SAGA: A Standard API for Grid Applications

The Simple API for Grid Applications (SAGA) [38] is an API standardization effort within the Open Grid Forum (OGF), an international committee that coordinates the standardization of Grid middleware and architectures. SAGA provides a simple, POSIX-style API to the most common Grid functions at a sufficiently high level of abstraction so as to remain independent of the diverse and dynamic Grid environments. SAGA has been referred to as the “MPI for Grid Programming,” in that is a simple, high-level programming abstraction that provides most required functionality. The interface defined by the SAGA specification is grouped as a set of functional packages, which we discuss in this section.

The SAGA landscape consists of three levels: 1) a high-level API; 2) SAGA Engine; and 3) adaptors that provide a specific package functionality to different distributions of middleware. Here we discuss the first two. The Louisiana State (LSU) group provides a native C++ Engine and well-documented and tested Python and C wrappers to the C++ Engine.

---

2 This section edited from material provided by Shantenu Jha, SAGA PI.
3.5.2.1 SAGA API Packages

As stated above, the SAGA specification defines interfaces for the most common Grid-programming functions, grouped as a set of functional packages.

Some key packages are described below:

File package. A file package provides methods for accessing local and remote file systems; browsing directories; moving, copying, and deleting files; setting access permissions; and zero-copy reading and writing.

Replica package. A replica package provides methods for replica management such as browsing logical file systems; moving, copying, deleting logical entries; adding and removing physical files from a logical file entry; and searching logical files based on attribute sets.

Job package. A job package provides methods for describing, submitting, monitoring, and controlling local and remote jobs. Many parts of this package were derived from the widely adopted DRMAA (Distributed Resource Management Application) API specification.

Stream package. A stream package provides methods for authenticated local and remote socket connections with hooks to support authorization and encryption schemes.

RPC package. An RPC (remote procedure call) package is an implementation of the GGF (Global Grid Forum) GridRPC API definition and provides methods for unified remote procedure calls.

SAGA achieves simplicity by limiting the scope to only the most common and important grid functionality required by applications. Major advantages arise not only from its simplicity, but also from its imminent standardization. Standardization confirms the interface’s derivation from a wide
range of applications through the collaborative efforts of a group whose output is endorsed by the broader community.

3.5.3  jGlobus Java API
Description: Java APIs for Grid Security Infrastructure (GSI), GridFTP client, and GRAM5 client.
Details on this component will be provided in a future version of this document.

3.5.4  Globus Online Java client API
Description: Java client API to the Globus Online REST APIs.
Details on this component will be provided in a future version of this document.

3.5.5  Globus Online Python client API
Description: Python client API to the Globus Online REST APIs.
Details on this component will be provided in a future version of this document.
4 Two Architectural Approaches

Before we proceed to describe the Service Layer interfaces, we need to spend some time describing the two different architectural approaches that are realized in those interfaces. The first, X-WAVE, is based on SOAP and related Web Services specifications. The second, the Web/Cloud approach, is based on the standards on which modern Web and Cloud systems are based.

4.1 Open Standards-Based Web Services Architecture

The set of operating system services has been fairly consistent during the last 30 years: process creation, process scheduling, process management, inter-process communication and synchronization, security, file systems, and memory management. This consistency reflects the realization that these are the core services and abstractions needed by applications. X-WAVE takes the notion of an operating system to the next level; as an operating system that uses host OS/HW pairs as the basic building blocks, X-WAVE is a meta-operating system. Just as does an operating system, X-WAVE performs process (job) creation, scheduling, and management; provides a secure operating environment; and provides file system-like data naming and management capabilities. Like Unix before it, X-WAVE provides a few simple abstractions for each of these processes that applications can manipulate.

This section focuses on defining the primitive services and schema that are combined to realize X-WAVE functionality.

X-WAVE is a standards-based Service Oriented Architecture based on Web Services. As such, it is defined by the set of XML schema used in interactions between services and by the set of interfaces implemented by Web Service endpoints (hereafter referred to simply as endpoints). Communication is via XML-encoded SOAP messages typically, but not necessarily, transported over http or https.

X-WAVE primitives can be divided into five basic categories: 1) Web Services infrastructure (communication, containers, and factories; naming and binding; discovery and reflection); 2) security; 3) directory and discovery services; 4) execution management; and 5) data management. These functional capabilities must be implemented in the context of quality-of-service attributes such as performance, reliability, and availability; usability; modifiability; and reusability.

The sidebar (above) lists several specific terms for these primitives, which we will reference in the following discussion.

<table>
<thead>
<tr>
<th>WS-Addressing EPRs</th>
<th>&quot;point&quot; to Web Service endpoints.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS-Naming EPRs</td>
<td>provide identity.</td>
</tr>
<tr>
<td>WS-Naming resolution services</td>
<td>enable naming transparencies.</td>
</tr>
<tr>
<td>Web Services containers and factory operations</td>
<td>instantiate WSEs.</td>
</tr>
<tr>
<td>RNS directory services</td>
<td>provide a human readable namespace.</td>
</tr>
<tr>
<td>JSDL documents</td>
<td>describe jobs.</td>
</tr>
<tr>
<td>GLUE2 - standardized information schema</td>
<td>for Grid infrastructures.</td>
</tr>
<tr>
<td>OGSAs-BES execution services</td>
<td>execute jobs.</td>
</tr>
<tr>
<td>OGSAs-ByteIO</td>
<td>are typed files and streams.</td>
</tr>
<tr>
<td>OGSAs-DMI</td>
<td>- standardized data movement interface.</td>
</tr>
<tr>
<td>WSI-BSP identity tokens</td>
<td>carry identity information such as a signed SAML token.</td>
</tr>
<tr>
<td>WS-Trust Secure Token Services</td>
<td>map identity tokens.</td>
</tr>
<tr>
<td>UR</td>
<td>is a schema for the standardized tracking or compute and storage resource usage.</td>
</tr>
<tr>
<td>RUS</td>
<td>is a standard interface to obtain and manage URs.</td>
</tr>
<tr>
<td>SRM</td>
<td>is a standard storage interface used widely around the world.</td>
</tr>
</tbody>
</table>
4.1.1 SOAP

As stated above, communication in Web Services is via XML-encoded SOAP messages. While strictly speaking, Web Services can support asynchronous communication or return results to a service other than the caller (e.g., DAGs), the vast majority of implementations use Web Services to implement traditional remote procedure call (RPC).

The interface description language for Web Services is WSDL (Web Services Description Language) [25]. WSDL is an XML-based language that describes the data types and interfaces of a set of Web Service porttypes. Tooling exists from many vendors and open-source communities that will automatically generate client stubs and server skeletons in a variety of languages when given a WSDL file as input.

A SOAP message is encoded as an XML document, consisting of an <Envelope> element, which contains an optional <Header> element, and a mandatory <Body> element. The <Fault> element, contained within the <Body>, is used for reporting errors. The body contains the name of the function to call and the actual parameters. The header contains information that the receiving service container may or may not utilize. Often, environment information such as the kind found in an Internet Inter-ORB Protocol (IIOP)context[18] is passed in the header.

In a Web Services RPC, the client calls a client stub routine, and the stub routine marshals the arguments into an XML/SOAP message, connects to the service (usually using http or https), sends the message, waits for the reply, parses the XML returned, and returns the values (if any) to the caller. The service waits at a well-known port for a socket connection request; reads the SOAP message; passes the message through a message-processing pipeline that includes signature checking, authorization, attachment(s) extraction; and ultimately calls the actual service requested. Once the service is complete, it sends a SOAP reply that goes back through another processing pipeline and finally back out over the socket. If the call fails, for example if the service is down or unreachable, the client stub can take appropriate action.

While synchronous RPC is the most commonly used distributed computing paradigm, it does not fit all problems well. In particular, it does not handle asynchronous events well, such as job completion/failure or the load change on a host. An entity that is interested in these asynchronous events could, of course, poll using an RPC, but polling can consume considerable resources (both network and compute), particularly if the poll interval is kept small so the client can know as soon as possible about a change. This problem has been handled for decades using event-based communication such as publish/subscribe, in which a client subscribes to an event that a service publishes. When the event occurs, the service sends a message to (or calls back) the client to notify that the event has occurred.

X-WAVE follows the Web Services Resource Framework model [37, 77]. All “things” of interest to the architecture are resources. Each resource:

- Has a WS-Addressing EPR address. The EPR may contain a unique ID that can be used to uniquely identify the resource. Some resource EPRs will have the ability to be rebound to point to a different address.
- Has an interface consisting of one or more WSDL interfaces.
- Has a (possibly empty) set of metadata known as resource properties.
- May have persistent state.
4.1.2 Containers and Factories: The Creation of Endpoints

Given the communication mechanism just described, the creation of endpoints is of critical importance. As is consistent with the Web Services general model, services will execute in containers as shown below. Service endpoints will come into existence, either as a side effect of a call on a service or explicitly via factory operations. Factory operations are operations whose purpose is to instantiate a new Web Service endpoint. The parameters to the factory operations may be service-type specific, or general-purpose factory operations may be used. When WS-Naming endpoint identifiers (EPIs) are used, it will be possible to instantiate an endpoint in a container with a specified EPI.

Figure 27: In a Web Services Architecture clients call services using XML-encoded SOAP messages conveyed over HTTP(S). Incoming requests are parsed and dispatched by a Web Services container and XML/SOAP engine (e.g., Apache Axis). Persistent state is stored in a relational database such as MySQL or Apache Derby.

4.1.3 Naming and Binding: The Naming of Endpoints

The goal of a naming and binding service is to provide mechanisms to support transparency with respect to location, migration, failure, replication, implementation, heterogeneity, and concurrency — for both humans and applications. In other words, application programmers should by default be unaware if the endpoint/service they are interacting with migrates, fails, or is replicated. They should be able to focus on what they are trying to do with the endpoint or service rather than what is going on behind the scenes. Users should also be able to use “names” that are meaningful to them (e.g., “/schedulers/metascrider1”) rather than complex URLs or XML documents.

Multi-level naming schemes have been employed in distributed systems over the years to simplify application development, deal with the complexity of the environment, and provide naming transparency [24, 48-50, 62, 63, 68, 69, 73, 80, 81]. In three-level naming schemes, human names (e.g.,
path names or registry queries) are first mapped to abstract names. Abstract names are location-

independent and intentionally opaque to the client. Before a client can communicate with a re-

source named via an abstract name, the abstract name must first be resolved or bound to an

address. The advantage of a three-level naming scheme is that the client can work with human

readable names while the infrastructure manages the binding logic.

We accomplish our naming and binding goals using a three-layer naming scheme.

There are three specifications of interest: RNS 1.1, WS-Naming, and WS-Addressing. They corre-

spond to the three naming layers. We will describe these beginning with WS-Addressing.

4.1.4 WS-Addressing

WS-Addressing [20] is a W3C specification that describes, among other things, a schema type called

an endpoint reference (EPR). Clients and services use these EPRs to identify target Web Service

resources by embedding information contained in the EPR into SOAP message headers. The EPR

schema includes fields for identifying the target address (URI) of the desired service, opaque

referencing information which services may use to further identify session data, and metadata

information which can be used by clients as hints that describe various aspects of the target Web

Service or Web Service resource. EPRs are the most widely used mechanism for referring to Web

Service endpoints.

4.1.5 WS-Naming

WS-Naming [40] was developed to address two shortcomings of WS-Addressing. First, EPRs cannot

be compared in any canonical way to determine if they refer to the “same” endpoint. Indeed, the

specification explicitly states that EPRs cannot be compared. Second, given the way many WS-

Addressing implementations work, an endpoint cannot migrate nor have a failover mechanism. To

understand why, we need to look at the wsa:Address field of the EPR. The wsa:Address field is a URI.

Technically, this URI could have a location-transparent abstract string; in practice, however, it does

not. Every WS-Addressing implementation we have encountered uses a URL with either an IP

address or a DNS hostname. Thus, the endpoint is pinned to a particular location at the time it is

minted. (There are tricks you can play with DNS to move endpoints around, particularly in a

machine room. However, these tricks do not work well across domains and break the advantages of

the cache.)

The authors of WS-Naming wanted a profile on WS-Addressing that would address these two

concerns and also be 100% compatible with existing practice regarding the use of WS-Addressing

endpoints. This is critical; EPRs that are WS-Names MUST be consumable by clients that are com-

pletely unaware of WS-Naming and that assume the wsa:Address field is a URL.

WS-Naming describes two extensibility profiles on the standard WS-Addressing specification,

whereby target service endpoints add additional information to their WS-Addressing EPR’s

metadata element. The addition consists of an endpoint identifier element (EPI) that serves as a
globally unique (both in space and time) abstract name for that resource and a list of zero or more

resolver EPRs.

Clients may use the embedded resolver information within a target EPR to call a resolver and

obtain a new binding (another EPR) for the target resource. For example, clients attempting to

communicate with stale or invalid endpoint references can use a resolver to obtain new, up-to-date

bindings. Note, however, that clients are free to choose how they obtain endpoint references —

they are not required to use the resolvers provided in the EPR. Any means for resolving EPIs or

stale EPRs may be tried at the discretion of the client.
WS-Naming, in conjunction with a resolver service, can be used to support migration and failure transparency as well as replica selection for improved performance [Grimshaw ConPE paper].

4.1.6 WS-Notification

Asynchronous event notification is a powerful capability that most grid middleware stacks implement. There are two different standards in the Web Services space, WS-Events (Microsoft) and WS-Notification (OASIS, IBM and others). The basic idea behind both is similar. XSEDE has settled on WS-Notification because it is the choice of many grid middleware stacks such as UNICORE 6, Genesis II, and GridSAM.

WS-Notification provides a traditional publish/subscribe functionality. Clients subscribe to particular topics of interest on an endpoint. Correspondingly, endpoints can publish (or raise) events. When an endpoint publishes an event, all subscribers to that topic are sent a copy of the notification message. The notification message contains an application-specific body.

Both a push and pull model of delivery can be supported within the specification family. This is important insofar as subscribers may be behind a NAT or firewall.

To address issues of scalability and permit a variety of different distribution modalities, the specification family includes the concept of a notification broker. Brokers distribute notifications on behalf of endpoints. Brokers can have different quality-of-service attributes such as reliable delivery.
Trees (or other topologies) of brokers can be constructed to improve the performance or reliability of notification delivery.

The use of a publish/subscribe mechanism is a powerful tool in building scalable applications. Without it clients must resort to using polling mechanisms to determine the state change of an endpoint. For example, they might poll an execution service to determine if a job has completed, poll a host resource to find out the load, or poll a directory or file to see if it has changed. With polling there is a tradeoff between the overhead of polling in terms of bandwidth and CPU resources and the timeliness of the information. More frequent polling provides more accurate information, but at the cost of increased overhead.

4.1.7 Reflection and Discovery: The Discovery of Endpoints

Reflection here refers to the ability to discover properties or attributes of endpoints; for example, the porttypes, security mechanisms, and the provenance of data. Examples in use include WSRF-RP [77] and WS-Metadata Exchange [14]. The OGSA WSRF Base Profile [37, 52] addresses selected WSRF-RP specifications, including the operation `getResourceProperties`, which returns an XML document containing the metadata associated with a resource. XSEDE services must support the OGSA WSRF BP.

```
<<Interface>>

WS-ResourceProperties

+vendorExtension : string
+vendorDiskDriveProperties : complexType
+getResourceProperty() : GetResourcePropertyResponse
+getMultipleResourceProperties() : GetMultipleResourcePropertiesResponse
```

**Figure 30. WS-Resource Properties interface.**

Examples of metadata depend on the “type” of resource and could include ownership, provenance, last update time, input files used in a simulation to produce the file, interfaces supported, and number of processors.

From a performance perspective, it is time consuming to constantly check whether endpoints support operations by calling `getResourceProperties` before you invoke them. As a performance optimization, the interfaces that the endpoint implements can be included in the metadata section of the endpoint EPR, eliminating an extra RPC.

In the Open Standards Based Web Services Architecture, metadata is logically kept at the endpoint; i.e., a `getResourceProperties` call on the endpoint will return the metadata. It is up to the particular implementation to decide where the information will be kept, e.g., in a database, in memory at the resource, or generated on the fly when needed.

As discussed earlier, discovery can be performed using RNS directory services. However, in some cases, a client may wish to search for an endpoint using a query over metadata. This is accomplished in XSEDE using registries and information services. There are as yet no registry/info-
services standards for dynamic metadata\(^3\), though query languages such as XPATH/XQuery have been developed. On Day One we will use the UNICORE 6 registry services.

A common issue for registries and information services is whether they use a “push” model or a “pull” model. Each method has its advantages and disadvantages. Pull models, since they require polling at some level, lead to a tradeoff between timeliness and accuracy of the data and overhead both on the network and at the endpoints. In a push model, information clients “subscribe” to changes in the state of the metadata and notifications are sent to subscribers when the data has changed. This improves both data “freshness” and reduces overhead, though it also can lead to increased overhead if the data is changing rapidly, and — because messages can be lost — can lead to lost data without a reliable notification system. Genesis II uses a pull model, while UNICORE-6 uses a push model. The XSEDE architecture is agnostic in this respect and can support both models.

### 4.1.8 OGSA WSRF Basic Profile

The OGSA WSRF Basic Profile [37] profiles the use of WS-Addressing, WS-Notification, and WS Resource Properties. The OGSA WSRF Basic Profile provides a standard set of basic interfaces on which clients can depend. UNICORE 6 and Genesis II support the OGSA WSRF Basic Profile.

### 4.1.9 Security: Securing Interactions Between Endpoints

The WS-Interoperability Basic Security Profile (WSI-BSP) [86] provides guidance for using the secure communication mechanisms (i.e., digital signature for integrity and encryption for confidentiality) and authentication tokens defined by the WS-Security (WS-S) [65] family of specifications. The authentication token types profiled include, among others, X.509, SAML, and username/password. WSI-BSP is supported by all major Web Service tooling and container providers.

WS-Trust [66] defines a *Security Token Service* interface. An STS is, in the general sense, a token acquisition service. The type and content of the returned token depends on the STS implementation and the identity tokens conveyed by the callee. STS services can be used to bridge authentication domains, as group or virtual organization verifiers, or as general identity services. One can easily imagine MyProxy [16, 53] services being wrapped with an STS interface.

<table>
<thead>
<tr>
<th>SecurityTokenService</th>
</tr>
</thead>
<tbody>
<tr>
<td>+requestSecurityToken() : RequestSecurityTokenResponse</td>
</tr>
<tr>
<td>+requestSecurityTokenCollection() : RequestSecurityTokenResponseCollection</td>
</tr>
</tbody>
</table>

**Figure 3.1.** WS-Trust Secure Token Service interface.

The WS-Security Policy [28] specification defines how security policies of WSEs can be described in a consistent manner. The client can use a WS-SecurityPolicy document to determine how it should interact with the service. The document states requirements regarding authentication tokens, integrity, and/or confidentiality, etc. The use of security policy documents is particularly important in federated environments where different organizations may require different mechanisms for secure interaction. Without a consistent way to describe the requirements, clients would either be unable to communicate with an endpoint or would have to try combinations blindly.

---

\(^3\)Universal Description, Discovery and Integration (UDDI) is a Web Services WSDL registry, but does not provide for arbitrary metadata, nor is it intended to be used for dynamic information.
"The EPR includes fields for identifying the target address (URI) of the desired service, opaque referencing information which services may use to further identify session data, and metadata information that can be used by clients as hints that describe various aspects of the target Web Service or Web Service resource. EPRs are the most widely used mechanism for referring to Web Service endpoints."

All EPRs used in Genesis II conform to the Secure Addressing profile [55-57]. Each EPR serves as a logical "invocation context" and includes within its definition:

- An embedded WS-SecurityPolicy document,
- An embedded X.509 certificate.

The security policy document conforms to the OGF Secure Communication 1.0 profile [56]. Figure 32 shows a sample EPR.

```xml
<wsa:EndpointReference>
  <wsa:Address wsu:Id='TheAddress'>http://www.example.org/some/path</wsa:Address>
  <wsa:ReferenceParameters wsu:Id='TheRefParams'>...</wsa:ReferenceParameters>
  <wsa:Metadata wsu:Id='TheMetadata'>
    <!-- This policy attachment applies to all actions on this endpoint -->
    <wsp:PolicyAttachment>
      <wsp:AppliesTo><wsp:URI>urn:wsaaction:*</wsp:URI></wsp:AppliesTo>
    </wsp:PolicyAttachment>
    <!-- Collection of policy alternatives -->
    <wsp:Policy>
      <!-- Server-authenticated TLS -->
      <!-- Acceptable message-level user-principal identities -->
      <wsp:ExactlyOne>
      </wsp:ExactlyOne>
    </wsp:Policy>
    </wsp:PolicyAttachment>
  </wsa:Metadata>
</wsa:EndpointReference>
```

Figure 32. Sample WS Secure Addressing EPR.

Clients use the WS Secure Addressing EPR to discover the security requirements of servers. The EPR also is used to extract X.509 public keys from X.509 certificates. In this way, clients can pre-delegate credentials to the X.509 holder of the key.

In the basic Web Services model, authorization decisions are ultimately enforced at the endpoint or by the endpoint container. Often container environments (such as Apache Axis, Web Sphere, or

---

4 All Genesis II generated EPRs conform to the WS-Addressing and Secure Addressing profiles. However, other systems may generate other EPR types. Genesis II can use those as well. For example, one could construct an EPR that "wraps" a GridFTP URL or a vanilla http URL. We have in fact done this for vanilla URLs.
Glass Fish) have a message-processing stack that includes authorization decisions. To change the authorization policy, one simply replaces the plug-in with a different module. The specification of the authorization module can be done either by a configuration file at container startup or dynamically at run-time, based on a database lookup.

4.2 The Web/Cloud Approach and XUAS

The commercial and consumer Web/Cloud market — as typified, for example, by the likes of Google, Facebook, Twitter, and Amazon — has, for the most part, converged over the past five years on a common architectural approach for defining interfaces to services. This approach, which we term here “Web/Cloud,” is based on a small set of protocol standards and methodologies, namely REST and HTTP, JSON and XML, TLS, and OAuth 2.0.

Two reasons frequently cited for the popularity of the Web/Cloud approach in the commercial and consumer market are (a) its relative simplicity (e.g., compare a conventional URL with the EPR in Figure 32), which means that Web/Cloud components can typically be accessed via a simple Web browser, with no need to install any local software; and (b) the loose coupling between client and server — consistent with long-held principles of wide area system design [72] — that is encouraged by the REST model. These two reasons also argue for the use of the Web/Cloud approach in the XSEDE architecture, at least in its user-facing components. In addition, designing the XSEDE architecture as a microcosm of the Web/Cloud ecosystem, embracing its standards, and integrating with its services, allows the NSF community to leverage the enormous investments being made in that space, and the resulting rapid pace of innovation.

These considerations lead to the incorporation within the XSEDE architecture of a set of components named the XSEDE User Access Services (XUAS, pronounced “Zeus”). In keeping with the Web/Cloud approach, XUAS deploys a set of hosted (“cloud”) services that provide XSEDE users with convenient interfaces to cyberinfrastructure capabilities. Access to those services is then provided via lightweight (“thin client”) interfaces.

In the rest of this section, we briefly introduce the core Web/Cloud standards and methodologies on which XUAS is built. XUAS is described in more detail in §6.

4.2.1 REST and HTTP

At the core of the Web/Cloud approach is REpresentational State Transfer (REST), initially defined by Roy Fielding in his Ph.D. dissertation [35]. As noted in Wikipedia [7], "REpresentational State Transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web."

The REST approach is embodied in HTTP. At its core is an architecture defined around resources that are named by URLs and interaction with resources through standard HTTP actions (POST, GET, PUT, DELETE), which roughly correspond to traditional Create, Read, Update, Delete (CRUD) actions, respectively, against a database. Richardson and Ruby's book [71] provides a good grounding on the practical principles of using a REST architecture. A rich set of capabilities in and around HTTP (e.g., DNS, load balancing, caching) allows for the construction of loosely coupled, global-scale systems such as the Web.

► Example: We use the Globus Transfer REST API to illustrate the use of REST and HTTP. Globus Transfer allows for the movement or synchronization of data between endpoints. Before requesting a transfer between two endpoints, a user must first activate each of them, which will (if necessary) initiate an authentication sequence to obtain a temporary proxy credential for that endpoint. The
Globus Transfer REST API models endpoints as resources with the following form, where NAME is the endpoint name:

```
/endpoint/NAME
```

Thus, a request to activate an endpoint called NAME is made by sending the following HTTP request to Globus Transfer:

```
POST /endpoint/NAME/activate
```

Requests are made to a base URL that incorporates version number information to allow for changes to the API. Thus, a user might generate the following request to activate the “ranger” endpoint as defined by the “xsede” account:

```
POST /endpoint/xsede#ranger/activate
```

which, with the standard base URL, corresponds to a POST to:

```
https://transfer.api.globusonline.org/v0.10/endpoint/xsede#ranger/activate
```

Having activated the required endpoints, the user can then make transfer requests. This process involves a GET to obtain a unique submission identifier and a POST to request a transfer. The body of the POST (not shown here) specifies the details of transfer request, and the body includes a unique TASKID identifier for the transfer request:

```
GET /transfer/submission_id
POST /transfer
```

The user can then monitor the transfer by generating GET requests of various sorts, with TASKID in the following representing the submission identifier obtained previously. For example:

```
GET /task/TASKID
GET /task/TASKID/subtask_list
GET /task/TASKID/event_list
GET /subtask/TASKID
GET /subtask/TASKID/event_list
```

### 4.2.2 JSON and XML

While REST and HTTP define how to name resources (URLs), messaging semantics, and message framing syntax, they do not prescribe particular methods of encoding message contents. Two primary message encoding schemes are used in the Web/Cloud Architecture: Java Script Object Notation (JSON) [27] and eXtensible Markup Language (XML) [21]. Many services support both, allowing the client to specify which it prefers. JSON has gained popularity due to its close affinity with Javascript, but XML has advantages, particularly for non-Javascript clients.

XUAS recommends that services support both JSON and XML message encoding.

► **Example:** The Globus Transfer REST API uses a GET on the /tasks/summary resource to request a summary of currently active tasks, with the response containing the descriptions of the tasks. The following is the JSON representation of a summary of a file deletion task:

```
GET /tasks/summary.json
{
    "submission_id": "14e7d3f2-af41-11e1-bf56-1231380b8963",
    "endpoint": "go#ep2",
    "recursive": false,
```
And the following is the equivalent XML representation:

```xml
GET /tasksummary.xml
<delete xmlns=http://www.globus.org/go/REST/representation length="2"
  xml:base="https://transfer.api.globusonline.org/v0.10/"
  submission_id="2d804070-af41-11e1-bf56-1231380b8963"
  deadline="2012-06-06 19:03:46+00:00"
  endpoint="go#ep2"
  label="example delete label"
  recursive="false"
  ignore_missing="false"
  interpret_globs="false"
  delete_item=path="/~/bashrc_copy_example"/>
</delete>
```

### 4.2.3 TLS (HTTPS)

The Web/Cloud Architecture uses Transport Level Security (TLS; often referred to, somewhat incorrectly, as SSL) [31] for server authentication, using *TLS certificates*, which are essentially equivalent to the *host certificates* traditionally used in the Grid community. TLS also provides for encrypted HTTP communication (HTTPS) between a client and server.

Most commercial and consumer Web/Cloud services do not support TLS client authentication, opting instead for message-level authentication, particularly OAuth. However, for backward compatibility with widely used Grid architectures, XUAS recommends that XSEDE services support TLS client authentication in addition to OAuth.

### 4.2.4 OAuth 2.0

The Web/Cloud ecosystem has largely converged on the use of OAuth for client authentication and fine-grain authorization of resource access. The OAuth 2.0 specification, a draft in IETF [6], is widely adopted by Google, Facebook, and many others. OAuth 2.0 supports both browser-friendly authentication and non-browser-based client authentication.

XUAS recommends the use of OAuth 2.0 for client authentication and fine-grained authorization of resource access against its services-layer REST interfaces.

OAuth uses similar approaches to the browser-friendly, privacy-protecting authentication provided by SAML-based [46] approaches such as Shibboleth [33]. However, SAML-based approaches have
seen little adoption in the broad Web/Cloud ecosystem outside universities and InCommon. SAML identity and attribute assertions could be integrated into XUAS, but it is an open question as to whether there are compelling use cases that would push XSEDE in that direction, particularly since it would be a substantial departure from the Web/Cloud ecosystem.

5 X-WAVE

The Level Three decomposition of X-WAVE follows the Level Two decomposition described earlier (§2). It is divided into execution management, discovery and information, identity services, data management, infrastructure, and help desk and ticketing services.

We present the Level Three decomposition as follows:

First, we present the service interfaces. The interfaces are implementation agnostic. These are divided into standard interfaces, and interfaces defined by either UNICORE 6 or Genesis II that are needed to understand the implementation and use cases later.

Second, we present the class definitions used in the two implementations described here. These are represented in UML using the <<provides>> and <<uses>> nomenclature.

Third, we present a package and deployment view. Here we describe the software packages made up by the implementations and show how those packages are deployed in XSEDE.

5.1 Services Layer Interfaces

5.1.1 Discovery & Information

5.1.1.1 Directory Services (RNS)

RNS [60, 61] is a simple Web Services directory service that maps strings to RNSEntries. An RNSEntry is an XML document that MUST contain a WS-Addressing EPR and may contain other arbitrary metadata information. The RNS interface has porttypes to add, remove, and look up entries.
Figure 33. The Resource Namespace Service and WS Iterator interfaces. The RNS interface is modeled on Unix directories. An RNS directory is essentially a table indexed by string entry names. Table elements, RNSEntries, contain an EPR (playing the role of a UNIX *inode*, and arbitrary metadata. The iterator interface provides a means to return extremely large lists of entries.

Note that the EPRs in the entries may “point” to just about anything, for example, another RNS endpoint, a gridFTP endpoint, a regular webpage, a OGSA-ByteIO, an instrument, a scheduler, or an execution container. Figure 34 shows how RNS can be used to construct directory trees where the leaves are Web Service endpoints. Note that RNS endpoints can correspond to “real” directories that point to “real” endpoints such as files or dynamically generated directories that are used to show some component state, such as the active jobs on a compute resource.
Figure 34. The role of RNS in directory construction. Boxes represent RNS directories. Smaller boxes with an RNS directory box indicate an RNS entry. Arrow lines indicate links, EPRs that "point" to other RNS directories or other resources. In this example, the root of the name space points to an RNS entry that contains RNSEntries that point to other RNSs. The Archives entry points to a list of archives at the centers. The Queues entry points to a list of three queues, and mainQ points to compute resources at DEISA, NCSA, and TACC.

5.1.1.2 Service Registries

X-WAVE consists of a wide variety of Web services in different technical areas of computing, data, security, and information. In order to have the contact information for these services, a registry service is necessary. A collection of endpoints using the WS-Addressing standard EPR can be realized using the WS-Service Groups specification of WS-RF [77]. This specification works well with the ecosystem of the whole WS-RF family of specifications (e.g., WS Resource properties and WS-Resource Lifetime).
5.1.2 Execution Management

For basic job management, X-WAVE uses OGF job management specifications and profiles. The OGSA 1.5 Architecture Description [47, 75, 78] and OGSA ISV Primer [64] provide good descriptions of Execution Management Services (EMS). Much of this section comes directly from the OGSA 1.5 Architecture Description.

Execution Management Services are concerned with the problems of instantiating, and managing to completion, units of work that may consist of single activities, sets of independent activities, or workflows. More formally, EMS addresses problems with executing units of work including their placement, "provisioning," and lifetime management. These problems include, but are not limited to, the following:

**Finding execution candidate locations.** The service needs to determine the locations at which a unit of work can execute given resource restrictions such as memory, CPU, and binary type; available libraries; and available licenses. The service also needs to consider what policy restrictions are in place that may further limit the candidate set of execution locations.

**Selecting execution location.** Once it is known where a unit of work can execute, the service must determine where it should execute. Making this determination may involve different selection algorithms that optimize different objective functions or attempt to enforce different policies or service-level agreements.

**Preparing for execution.** Just because a unit of work can execute somewhere does not necessarily mean it can execute there without some setup. Setup could include deployment and configuration of binaries and libraries, staging data, or other operations to prepare the local execution environment.

**Initiating the execution.** Once everything is ready, the execution must be initiated and other related actions (such as registering it in the appropriate places) carried out.

**Managing the execution.** Once the execution is started, it must be managed and monitored to completion to deal with potential job failures or failure to meet its agreements. There must be protocols regarding whether the job should be restarted in another location (if it fails) or whether checkpointing (a process that can ensure restartability) should be performed. In general, it must be decided whether the execution occurs as a part of some sort of fault-detection and recovery scheme.

The solution to these five problems consists of a set of services that decompose the EMS problem into multiple, replaceable components.

5.1.2.1 Job management

The Job Manager (JM) is a higher-level service that encapsulates all of the aspects of executing a job or a set of jobs from start to finish. A set of jobs may be structured (e.g., a workflow or dependence graph) or unstructured (e.g., an array of non-interacting jobs). The JM may be a portal that interacts with users and manages jobs on their behalf. The JM is the only intentionally unspecified, non-
standard component of EMS, a condition that encourages the development of a plethora of different styles and capabilities.

The JM may interact with an Execution Planning Service, the deployment and configuration system, containers, and monitoring services. Further, it may deal with failures and restarts, schedule jobs to resources, collect agreements and reservations.

The JM is responsible for orchestrating the services used to start a job or set of jobs, by, for example, negotiating agreements, interacting with containers, and configuring monitoring and logging services. It also may aggregate job resource properties from the set of jobs it manages.

Examples of JMs include:

- A “queue” that accepts, prioritizes, and distributes “jobs” to different resources for computation. The JM tracks jobs; may prioritize jobs; and may have QoS facilities, a maximum number of outstanding jobs, and a set of service containers in which it places jobs. The Genesis II grid-queue is an example of this type of JM.
- A portal that interacts with end users to collect job data and requirements, schedules those jobs, and returns the results. We see today’s existing community gateways and portals as operating this way.
- A workflow manager that receives a set of job descriptions, QoS requirements, their dependence relationships, and initial data sets (think of it as a data flow graph with an initial marking), and schedules and manages the workflow to completion — perhaps even through a number of failures. The UNICORE 6 workflow engine is an example of this type of JM.
- An array job manager that takes a set of identical jobs with slightly different parameters and manages them through completion.
- A deadline manager that takes jobs annotated with QoS metrics such as start-time, deadline, reliability (completion probability), and budget and selects a set of resources that will meet the requirements on the user’s behalf. If no schedule exists, the user is notified.

XSEDE has three job managers as of this writing: 1) a simple job manager that submits the job to a BES or grid-queue and waits for it to complete, giving periodic status reports; 2) a grid-queue job manager that implements a priority queue with resource-matching that distributes jobs out to different OBSA-BES hosts; and 3) a workflow JM from UNICORE 6.

To realize the execution management functionality just described, we use a number of OGF specifications. Below we briefly introduce each one and show how they are used together.

5.1.2.2 Job Submission Description Language

**JSDL** (Job Submission Description Language) [12] documents are XML documents that describe a job: its resource requirements (such as memory, number and type of CPUs, and supported libraries); its input and output files (where they can be found, file access protocols to be used when staging data in and out); and the parameters to be passed to the application. If the application is not installed for a particular execution environment, it must first be installed. Often this is accomplished by staging-in the application as well as the input data files. JSDL files are given to execution services to execute the described job.

**JSDL Activity Instance Schema** defines an XML schema for logging the progress of JSDL job instances through their lifetimes, from instantiation through scheduling to placement, etc. The goal is to have a standard porttype and schema with which services that handle a job can log job-related
events. These capabilities are critical in unwinding what happened during failed job executions and for post-execution audits. (This specification is nearing completion in OGF. Once complete, our existing and similar mechanism will be modified to be compliant with job history specifications defined in OGF.)

5.1.2.3 OGSA Basic Execution Services

OGSA-BES (Basic Execution Services) [39] is a simple interface for creating new jobs, monitoring them, managing them throughout their lifetimes, and providing information useful for making scheduling decisions. Note that BES endpoints can “wrap” a variety of different back-end execution management systems, including fork/exec (in Unix), spawn (in Windows), PBS, LSF, SGE, or Torque queuing systems, or hierarchical collections of other BES endpoints.

```
<<Interface>>
BES-FactoryPortType
+isAcceptingNewActivities : boolean
+commonName : string
+longDescription : string
+totalNumberOfActivities : int
+activityReference : EPR
+totalNumberOfContainedResources : int
+containedResource : anyType
+namingProfile : URI
+BESExtension : URI
+localResourceManagerType : URI
+resourceName : string
+operatingSystem : string
+CPUArchitecture : string
+CPUCount : double
+CPUSpeed : double
+physicalMemory : double
+virtualMemory : double

createActivity(activityDocument : XML) : CreateActivityResponse
getActivityStatuses(activityIdentifier : EPR []) : GetActivityStatusResponse []
terminateActivities(activities : EPR []) : TerminateActivityResponse []
getActivityDocuments(activityIdentifiers : EPR []) : GetActivityDocumentResponse []
getFactoryAttributesDocument() : BESResourceAttributesDocument

<<Interface>>
BES-ManagementPortType
+stopAcceptingNewActivities()
+startAcceptingNewActivities()
```

Figure 36. OGSA Basic Execution Services interfaces.

On top of the JSDL and OGSA-BES specifications, the HPC Profile Group inside OGF has defined a number of specifications and profiles on existing specifications that further aide in interoperability.

5.1.2.4 HPC-BP (High-Performance Computing Basic Profile)

HPC-BP (High-Performance Computing Basic Profile) [32] defines a simplified application element that can be used inside of JSDL documents to more easily annotate how a sequential application should be run (executable name, command-line arguments, etc.).

5.1.2.5 HPC-FSE (File-Staging Extensions)

HPC-FSE (File-Staging Extensions) [84] expands on the normal JSDL Data Staging elements to standardize data staging protocols that can be used to copy data in and out of an application run. XSEDE will support the file staging protocols rns, http, https, ftp, gridftp, and scp for staging files in and rns, ftp, scp, gridftp, and mailto for staging files out.
5.1.2.6 Activity EndPoint Profile

The Activity Endpoint Profile is a profile on the EndPointReference returned by an OGSA Basic Execution Services CreateActivity call. It defines porttypes that the endpoint must support, values that MUST be returned from calls, and values that MAY be returned. The goal is to provide a uniform mechanism to support requirements identified by the Production Grid Interoperability Working Group of the Open Grid Forum.

5.1.2.7 BES Directory Profile (BDP)

The BES Directory Profile (BDP) is a profile on OGSA Basic Execution Services 1.0 that provides a Unix directly-like interface to OGSA BES endpoints that allow the client to examine and modify BES configuration information, as well as list, create, and terminate activities. The profile defines no porttypes; rather, it simply describes what directory elements must exist in a BDP-compliant implementation and the semantics of interacting with those entries. The goal is to provide a simple uniform mechanism to support requirements identified by the Production Grid Interoperability Working Group of the Open Grid Forum. The mechanism described for the BDP is independent of the OGSA BES specification. Thus, changes in the OGSA BES specification in the future will not impact the usefulness of the BDP.

A number of additional resource selection services have been specified but not yet included in implementations as separate services. The capabilities these services provide are embedded within existing systems such as UNICORE 6 and Genesis II, though they are not exposed.

5.1.3 Data Management

Now that we have discussed the primitive functionalities, we need to consider another element of the service layer, data management. There are three data-access modalities we must consider: stage-in/stage-out mechanisms as required by large data jobs\(^5\), access to local data sets, and direct access to remote data resources via file-system-like capabilities. Stage-in/out is supported as in the HPC-FSE or as separate “data transfer” jobs. Local data access will be supported by the local file system. Direct access to remote data (including possibly for staging) will be provided via Web Services and the XWFS. In the remainder of this section we consider only remote data access, as staging is covered above by the HPC-FSE.

When we speak of data management, we refer to the ability to create, destroy, name, and read/write random-access and stream-oriented files. To support this functionality, XSEDE will support OGSA-ByteIO to represent random-access and streaming files. RNS will be supported as the primary naming mechanism. The RNS specification reveals named entities in a namespace, while the ByteIO specification enables manipulation of any bytes of data associated with these named entities.

Nearly every XSEDE grid resource available to the user will exist as a named path in a grid-wide RNS space. (This includes everything from file and directories to execution containers, queues, running applications and even non-grid web sites.) Because RNS paths can refer to any relevant grid (or web) service, one achieves a hierarchical (or directed-graph, file-system-like) organization by allowing a named mapping inside an RNS resource to indicate another RNS resource (in much the same way as an entry inside a file system’s directory may be another file system directory). A distributed namespace emerges, which, along with human organizational conventions, provides a

\(^5\) For jobs that have large input or output files, the data really needs to be “close” to the execution machine before the job starts or the application will spend a significantly greater amount of time on I/O than would be the case when the data is local.
complete and familiar picture to the end user. Despite the fact that the implied directory structure exists in a distributed grid that could span computers and countries, the familiarity of the file system view makes it easier for the end user to learn and navigate the new environment.

5.1.3.1 OGSA-ByteIO (GFD.87)

The ByteIO [58] specification describes a standard way of handling the transfer of data associated with grid endpoints, e.g., reading and writing. ByteIO has two separate porttypes: random access and streamable.

---

**Figure 37. OGSA ByteIO interfaces for random-access files and streams.**

ByteIO is not limited to presenting file information, despite its intentional design as a file-like resource. Implementers may choose to use ByteIO to present a file-like interface to any source or sink of data (such as a memory region in a running simulation or a streaming instrument). Similarly, RDBMS tables can be given ByteIO interfaces and the contents read and written as a normal text file.

Streamable ByteIO resources can be used to implement secure pipes and can be attached to stdout/stderr in running jobs, to name just two uses. Given the success and familiarity of pipes in Unix, we expect this to be useful.

ByteIO implementations have been written for a variety of back ends to accomplish a variety of tasks: to create and manage files on a backend data store; to provide access to existing files in a Unix, MacOS, or Windows file system; to represent tables in a RDBMS as flat files; to dynamically generate data that represents the internal state of a service; and (as a prototype) to read data from the CERN LFC [74].

Given such precedents, one can imagine implementations that wrap archival data management systems, the memory address space of a running process, or entries in an accounting database. One can similarly imagine ByteIO “files” that are transformations of one or more other data sources (including ByteIO files) and when “read,” execute a data reduction or processing step that generates the desired view.

5.1.3.2 GridFTP

See §7.1.1.1 for a description of the GridFTP interface.
5.1.3.3 Storage Resource Manager (SRM)

The Storage Resource Manager (SRM) is a broadly adopted standard interface for storage systems [SRMREF] of OGF. It includes a Web services protocol as well as functionality such as dynamic space allocation and file management on shared storage systems. SRM is designed to call on transport services to bring files into their space transparently and provide effective sharing of files. There are approximately seven different interoperable implementations (e.g., dCache, Disk Pool Manager, Berkeley Storage Manager (BestMan), StoRM). Many of the SRM adoptions are used as part of the European Grid infrastructure (EGI) or the Open Science Grid (OSG). Scientific uses are primarily those from LHC experiments, but SRM systems are also used across many different scientific domains. Many middleware implementations provide SRM clients to access LHC datasets or data from other communities.

Figure 39. SRM interface from GFD129 [76].
5.1.3.4 OGSA-Data Movement Interface (DMI)

The OGSA-Data Movement Interface (DMI) is a high-level interface that abstracts the wide variety of transfer mechanisms used to transfer files from source to sink. An implementation is available in UNICORE6 and several others.

[DMIREF] GFD187

5.1.4 Infrastructure Services

See §4.1 on Web Service standards.

5.1.5 Identity

X-WAVE uses the WS-Trust family as described in §4.1.9 to mint identity tokens. Identity (and delegated identity) is conveyed via authentication tokens defined by the WS-Security (WS-S) [65] family of specifications and between any client/server pair during the TLS handshake. The authentication token types profiled include, among others, X.509, SAML, and username/password. WSI-BSP is supported by all major Web Service tooling and container providers.

5.1.6 Accounting & Allocation

The OGF Usage Record Format (UR) [41] defined a schema for tracking resource consumption. It enables resource usage tracking of compute XSEDE resources, and new versions of the UR record also are able to perform usage tracking of storage XSEDE resources. This functionality is crucial for X-WAVE in order to perform accounting beyond each individual service provider and to ensure that usage record tracking is done in the same way across XSEDE. This in turn enables statistics for project management in terms of which resource capabilities are used in which way and when storage capacities become problematic. Figure 40 lists an example of the UR schema with some values arising from a computational job.

![Figure 40. UR instance example of a computational job.](image)

The Resource Usage Service (RUS) standard interface is used to manage (remotely) UR instances. The interface also enables the retrieval of record histories to check the origin and changes performed on specific UR instances. Newer versions implement the WS-Iterator to ensure scalability of records that can reach from only a few on HTC resources to thousands on HPC resources.
5.1.7 Help desk & Ticketing
Uses the existing XSEDE help desk system.

5.1.8 Genesis II Specific Interfaces
There are a number of interfaces that are specific to the Genesis II implementation. These are described below.

5.1.8.1 GridQueue

The GridQueue interface provides job queue-like interfaces familiar to users of queuing systems, e.g., submit, kill, etc. It also provides interfaces to get the job history; where the queue has tried to execute the job; and a detailed set of events including files staged, when, how many bytes, and the exit code of the job.

The grid queue interface is described in detail in 5.2.1.3.
5.1.8.2 WorkflowEngine

```java
<<Interface>>
WorkflowEngine

+submitWorkflow(dagDefinition : WorkflowDAGType) : string
+listWorkflowDags(mineOnly : boolean) : ReducedWorkflowDAGType []
+getWorkflowStatus(workflowTicket : string) : dagEnum
+killWorkflow(workflowTicket : string) : dagEnum
+removeWorkflow(workflowTicket : string) : boolean
+getWorkflowJobs(workflowTicket : string) : ReducedJobType []
+getWorkflowDag(workflowTicket : string) : WorkflowDAGType
+holdWorkflow(workflowTicket : string) : dagEnum
+resumeWorkflow(workflowTicket : string) : dagEnum
+cleanupWorkflow(workflowTicket : string) : boolean
```

Figure 43. The DAGMAN Workflow Engine interface.

DAGMAN [2] is a simple file format for representing workflows as directed acyclic graphs developed by the Condor team at the University of Wisconsin. Vertices in the graphs can be execution scripts used by Condor to run one or more jobs or they may be graph files themselves, allowing a recursive expansion of program graphs.

Like the GridQueue, the DAGMAN workflow engine in Genesis II files the role of a "Job Manager" in the execution management services architecture described in Section 5.1.2. The DAGMAN workflow engine in Genesis II uses the DAGMAN format, but instead of Condor scripts, the user can specify either a JSDL file or a DAGMAN graph file.

DAGMan’s syntax is simple: The entire language consists of only about a few dozen keywords, each with strict semantics. Each node in a workflow is a JOB, DATA task, or SUBDAG, and may include a PRE- and/or POST-SCRIPT; dependencies are specified as PARENT/CHILD relationships. Subdags, or smaller workflows embedded inside the total workflow, may be SPLICE-d into the parent, resulting in a single monolithic workflow structure, or executed EXTERNAL-ly with a distinct instance of the DAGMan.

The graph is not required to be fully connected; a single DAG file may define multiple independent workflows to be run simultaneously or even a list of completely independent jobs.

Should a node in the graph fail to execute correctly, DAGMan may be configured to retry the job up to a given number of attempts or the workflow can be halted and resumed at a later time. To enable this latter option, DAGMan will output a Rescue DAG file when a workflow fails. This file is mostly a copy of the original submission file, with a DONE annotation on each task that finished successfully. When this file is resubmitted to the workflow engine, the workflow will be resumed where the nodes had failed on the previous attempt, the assumption being that the problem will have been rectified based on information found in log files or other external resources.

The XSEDE DAGMAN workflow engine is designed as an emulator of DAGMan, which runs on the Genesis II platform. We implement this emulator as a new service that uses the existing infrastructure of a XSEDE Grid. The files that are used to submit workflows to the service reside in the GFFS Namespace (the Grid “file system”), and the jobs are scheduled on the Grid’s Queues and executed using the Grid’s BESs.

The primary functions of our service are threefold:
• Parse the submission file into a graph of interdependent nodes, which represent jobs to execute.
• Submit these jobs to the Grid resources in the order specified by the graph dependencies.
• Monitor running jobs to ensure successful completion before submitting additional jobs.

Once a workflow submission file has been parsed and accepted by the service, execution of jobs begins immediately. Each job in the graph is analyzed to determine if it has outstanding dependencies on other jobs, and those jobs with no dependencies are sent to the Grid’s queue. The queue handles resource matching and job placement and management.

The service will then periodically query the queue for the status of any jobs that belong to that workflow. If a running job completes, the graph is analyzed again to find any newly available jobs and these are sent to the queue. If a job fails, the workflow marks that node and the dag itself as an error and aborts execution.

Once the service begins execution, the user may query the service for the status of the workflow or issue commands to control the execution. Several command-line tools are available for interacting with the service:

• wsub – to submit new workflows to the service.
• wstat – to check the status of a workflow.
• wlist – to see a list of the workflows currently being managed by the service.
• wjobs – to see a list of the jobs for a given workflow, including their statuses and the job ticket used to reference the jobs in the queue.
• wclean – to “clean up” a dag, either during or after execution, which removes the completed or erroneous jobs from the queue.
• whold – to temporarily pause execution of a workflow. (Currently running jobs will continue, but no new jobs will be scheduled.)
• wresume – to resume a paused workflow.
• wkill – to abort the execution of a running workflow.
• wrescue – to output a “rescue DAG” for a workflow, which can be sent directly to the manager to retry execution at a later time.
• wrm – to remove a workflow from the service’s management.

Further details about each tool are available in the manual on the Genesis II developer’s wiki.

5.1.8.3 Ping

Ping is useful for testing whether a resource is alive and functioning using a common and trivial interface. If the ping returns with the same content it was sent, then the grid resource is alive and functioning.

5.1.8.4 VCGR-CreatePortType

Vcgr_create is a generic factory interface used for creating new resources. It takes one parameter, an XML construction properties document. The format of the document varies depending on which type of component the operation is invoked. Different component types then implement the interface differently. For example, the construction properties for a GeniiBESPortType indicates whether
the BES is a fork/exec simple BES or a BES that sends jobs to a queuing system such as LSF. In the case of a queuing system, it also carries information on the queue settings, file systems to use for staging files, etc.

5.1.8.5 CreateFile

Create file is a specific factory operation for creating RandomByteIO files in an RNS 1.1 directory. The RNS 1.1 specification indicates how to create new directories within directories but is silent on how to create files. We therefore created our own factory operation.

![CreateFile interface](image1)

![Ping interface](image2)

![VCGR-CreatePortType interface](image3)

Figure 44. CreateFile, Ping, and VCGR-CreatePortType interfaces.

5.1.8.6 GeniiBASE

![GeniiBASE interface](image4)

Figure 45. The GeniiBASE interface implements the WS-RF, WS-Notification.

All Genesis II resources implement the GeniiBASE class, which in turn implements WS-RF, WS-Notification, Ping, and VCGR_Create interfaces. This is so that all Genesis II resources provide the same basic functionality.
5.1.8.7  GAML Delegated Identity Chains

GAML certificates are SAML-like, Java serializable, signed assertions. There are methods to sign, add assertions, get assertions, and check the signature and trust chain. Java SE security classes are used to implement GAML signing, signature checking, etc.

```java
/** * A serializable statement*/
public interface Attribute extends Externalizable {
    /** Checks that the attribute is time-valid with respect to the supplied date 
        and any delegation depth requirements are met by the supplied delegationDepth. */
    public void checkValidity(int delegationDepth, Date date) throws AttributeInvalidException;

    /** Returns the identity of the attribute asserter */
    public X509Certificate[] getAssertingIdentityCertChain();
}

/** A signed attribute that can be authenticated via holder-of-key */
public interface SignedAssertion extends Externalizable {
    /** Returns the primary attribute that is being asserted*/
    public Attribute getAttribute();

    /** Returns the certificate chain of the identity authorized to use this assertion 
        *(same as the asserter)*
    */
    public X509Certificate[] getAuthorizedIdentity();

    /** Validate the assertion. It is validated if all signatures successfully 
        *authenticate the signed-in authorizing identities: an integrity-check.*/
    public void validateAssertion() throws GeneralSecurityException;
}

/** A delegated attribute. The pairing of an existing signed assertion with the identity of a delegate */
public interface DelegatedAttribute extends Attribute {
    /** Returns the identity needed to authorize the delegatee (i.e., the asserter 
        *of the outermost wrapped signed assertion) *
    */
    public X509Certificate[] getAuthorizedIdentity();

    /** Returns the delegatee identity */
    public X509Certificate[] getDelegateeIdentity();

    /** Checks that the attribute is time-valid with respect to the supplied date 
        *and any delegation depth requirements are met by the supplied 
        *delegationDepth.*
    */
    public void checkValidity(int delegationDepth, Date date) throws AttributeInvalidException;

    /** returns the signed assertion component */
    public SignedAssertion getSignedAssertion();
}

/** A signed, delegated attribute assertion */
public interface DelegatedAssertion extends SignedAssertion {
    /** Returns the identity of the original attribute asserter */
    public X509Certificate[] getAssertingIdentityCertChain();

    /** Returns the identity of the delegator */
    public X509Certificate[] getDelegatorIdentity();

    /** Returns the identity authorized to use the assertion (the delegatee)*
    */
    public X509Certificate[] getAuthorizedIdentity();

    /** Checks that the assertion is time-valid with respect to the supplied date 
        *and any delegation depth requirements are met by the supplied delegationDepth.*
    */
    public void checkValidity(int delegationDepth, Date date) throws AttributeInvalidException;

    /** Unwraps the delegated assertion by one layer */
    public SignedAssertion unwrap();
}
```

Figure 46. The Java interface for signed delegated assertions.
NOTE: Use of GAML is deprecated in the architecture and is being replaced by signed SAML certificates. A set of classes that perform the same actions on signed SAML certificates is under development.

5.1.9 UNICORE 6 Specific Interfaces
Details on this component will be provided in a future version of this document.

5.2 Services Layer Component Types (Implementation Specific)
The next function to consider in the service layer is that of interface composition. One of the nice properties of the Web Services model is that porttypes define interfaces rather than implementations, and interfaces can be combined in arbitrary ways without the mental gymnastics required in a purely object-oriented world. This is a powerful architectural feature that we will leverage in a number of ways.

We begin with the composition of RNS with other porttypes. The RNS interface gives us a consistent way to iterate over, add to, delete, and look up lists of named things.

5.2.1 Execution Management
5.2.1.1 GeniiBESPortType
For example, if we combine RNS with OGSA-BES, we can define a list operation to return the list of jobs in the BES. The job name elements of the JSDL can be used as the directory entry name (or a made-up string if there is no job name), and the EPR element can be the EPR of the corresponding BES-Activity. A directory listing in the BES will thus provide a list of jobs. If the BES-Activity also supports RNS and ByteIO, one could drill down within the job entry list and expand that directory. A convention can be used wherein the listing of a BES Activity gives entries for a status file, the JSDL for the job, and another directory pointing to the working directory of the job. Users could execute the cat command on the status file to see the state of the job and any errors detected.

Similarly the use of RNS and ByteIO gives the GeniiBESPortType a RESTian capability. Users can cp or cat a JSDL file into the BES’s submission point (an RNS directory "in" the BES) to create a new activity. The activity can then be monitored and managed via POSIX IO functions.
Figure 47. The GeniiBESPortType combines several different interfaces. The RNS and ByteIO interfaces give users the ability to start and manage jobs using file system operations.

5.2.1.2 BESActivityPortType

This profile is compliant with the OGF Activity Endpoint Profile (EAP) currently under development. The Activity Endpoint Profile is a profile on the EndPointReference returned by an OGSA Basic Execution Services CreateActivity call. It defines port-types that the endpoint must support, values that MUST be returned from calls, and values that MAY be returned. The goal is to provide a uniform mechanism to support requirements identified by the Production Grid Interoperability Working Group of the Open Grid Forum.

The Activity Endpoint Profile is a profile on the EPR returned from CreateActivity. The profile

i. specifies that the OGSA-BES returns an EPR that implements the RNS 1.1 OGSA-WSRF Basic Profile 1.0 [GFD.172];

ii. defines WS Addressing metadata fields that MUST be present in the EPR of the activity;

iii. defines optional WS-Notification subscriptions;

iv. defines a set of required RNS entries and optional RNS entries returned from the RNS lookup operation on AEP compliant endpoints;

v. defines a set of resource properties exposing the capability, e.g., status;

vi. and defines an activity port-type.
5.2.1.3 QueuePortType

The QueuePortType implements a FIFO meta-scheduling service across multiple OGSA-BES endpoints. Once instantiated, each grid queue is configured with a set of resources that implement the OGSA-BES interface (including other QueuePortType instances, permitting hierarchies). For each BES resource, the number of slots is specified. The queue is permitted to send up to slots activities (jobs) to the BES. Thus, if five BESs were configured for a queue and each was configured with five slots, the GridQueue could have 25 active jobs at a time.

QueuePortType instances periodically poll their configured BES resources to fetch the BES Factory Attributes, information such as the CPU type and number, memory available, operating system, etc. The information is stored in an internal database and is used when matching activities to resources.

The submitjobs function accepts both single JSDL documents as well as a set of JSDL documents. JSDL document profiles supported include Posix and Parameter Sweep. Using the Parameter Sweep profile, tens of thousands of jobs (activities) can be submitted with a single JSDL. The QueuePortType implementation has been tested with more than 10K jobs, and we believe it can scale far larger. (The only problem as you increase the number of jobs is that it takes longer and longer to list all of the jobs and can exhaust the memory of the client when listing a large number of jobs). Note that jobs are only purged from the queue after a configurable amount of time (currently one month). This allows users to examine job histories and logs after the jobs complete. Jobs may be purged manually at any time.

Scheduling. QueuePortType instances implement priority-based queues subject to resource matching. (In other words, if the next job to schedule does not match with a schedulable resource, it is deferred until a matching resource is available.) To provide a degree of fairness within a priority level, jobs are scheduled round-robin between users. For example, if user A has 5,000 jobs in the

---

Figure 48. The QueuePortType implements GridQueues, OGS BES, RNS, and ByteIO. Jobs can be started and managed using the Web Services interfaces and through file operations via the RNS and ByteIO interfaces.
queue, and user B has 5 jobs in the queue, as resources become available for execution, the next job to execute will alternate between user A and B until all of B's jobs are running.

**Notifications.** QueuePortType instances subscribe to BES state-change notification on JSDL activities. Because not all BESs support notifications and notifications can be lost (i.e., they are not reliable), the QueuePortType periodically polls activity status if it has not received a notification. The default period is five minutes. To prevent overloading the server and networks, a configurable maximum number of poll operations are performed concurrently.

**Reliability.** All state-changing operations on the QueuePortType, e.g., job submission, update of activity state via a poll or notification, are persisted in the transactional database. Thus, if the container or host fails, then no information is lost and the QueuePortType will pick up where it left off. No jobs have ever been lost (though many have failed).

**Retries.** Activities can fail for many reasons, not all of them the "fault" of the activity. For example, a file to be staged may not be available, the administrator at a site may kill the job, the site might lose power, the disk may fill up, etc. The QueuePortType will restart each job up to a configurable number of times to mask failures that are not the job's "fault." The current default is five times. The QueuePortType attempts to determine from the failure type whether the "fault" is the job's or the local environment. If it can be positively determined to be the local environment's fault, then the job restart is not counted against the number of retries.

**Steps taken in a GridQueue when host fails while running a job:**

1) The grid queue GQ will detect the failure via repeated failures to complete the get-status call. At that point, GQ will:
   a. Destroy the job. Place a “destroy” call to the BES to destroy the activity into the container “persistent outcall” database. The call is there to clean up job state when the BES recovers. Attempts will be made for one month to clean up the job.
   b. Reschedule the job. Place the job back into the queue to be scheduled on another resource.
   c. The job will be rescheduled up to a configurable number of times. The default is five.
5.2.2 Data Management

5.2.2.1 LightWeightExport

The LightWeightExport interface implements the RNS 1.1, WS-Iterator, OGSA-ByteIO, and CreateFile interfaces and provides a mechanism to map existing directory-based file systems into the GFFS.

The LightWeightExport service implements the RNS and ByteIO interfaces and provides a mechanism to map existing directory trees in existing file systems into the GFFS. The existing file system must be accessible from the container where the service is running. For example, a LightWeightExport service running on a Windows desktop can export (share) directories on the local C: and D: drives, as well as on network file systems that have been mounted. Similarly, files stored in a Luster or NFS file system can be exported from a Linux machine, or a tape archive that is visible via a hierarchical storage management system also can be mapped into the GFFS.

Note that LightWeightExport does not copy data into the GFFS; it merely provides an alternative mechanism to access the data. Thus, when a file is modified locally via the local file system, those modifications are visible immediately in the GFFS. Similarly, the effects of modifying, creating, or destroying files via the LightWeightExport are immediately visible via the local file system.

The syntax for creating a LightWeightExport from the grid command line is:

```bash
eexport --create { --url <export-service-url> | <export-service-hint> }

   [--svn-user=<svn-username> | --svn-pass=<svn-pass> | --svn-revision=<svn-revision>]

<local-path> [new-rns-path]
```

Thus, to export the c:\XSEDE-testdir directory on my local Windows 7 desktop into the GFFS at the path /home/andrew/x-testdir, assuming I have linked my Genesis II container into the path /home/andrew/AG-DT, I would type:
export --create /home/andrew/AG-DT/Services/LightWeightExportPortType c:/XSEDE-testdir
/home/andrew/x-testdir

Access control. All access to local files via the LightWeightExport is done via the local file system using the user ID and permission of the container in which the LightWeightExport is running. Thus, in order to read a file via the GFFS and the LightWeightExport, the LightWeightExport must have read permission on the file and read/execute permission on the path to the file in the local file system, similarly for write, create, and destroy. When running on users’ desktops or private machines, this is rarely a problem. When running with a shared container such as at the SPs or on departmental file servers (which are almost always Unix-based), we recommend that extended access control lists be used. Extended access control lists are supported in Linux, Mac, and Windows 7. The idea is to provide the desired degree of access to the particular directories and files for the container user. (We sometimes call this the GFFS User.) In Linux this is done in two setfacl commands shown here from the script setpermissions.sh (in the /bin directory of the GFFS.) $griduser refers to the user ID of the GFFS-User, and $DIRPATH refers to the directory to be shared.

#Set ACLS recursively on directory
setfacl -R -m u:$griduser:rwx "$DIRPATH"

#Set default ACLs recursively
#so future files created with same acl
#allowing griduser and current user full access to files/directories
setfacl -R -d --set u:$griduser:rwx,u:$USER:rwx "$DIRPATH"

Notifications. To improve performance, RNS and ByteIO clients often subscribe to update notifications in order to cache data for extended periods of time. This is supported, for example, in the EnhancedRNSPortType (5.2.4.2) implementation. This easily can be accomplished for resources that only can be updated via Web Services. Unfortunately this is not the case for LightWeightExports — the files and directories can be updated via the local operating system using local file system operations that are not visible to the LightWeightExport implementation. Some file systems (e.g., Luster) do provide notifications, but, in general, file systems do not. Therefore the current LightWeightExport implementation does not support update subscriptions. This requires clients to periodically poll or invalidate their caches. It has been suggested by operations stakeholders (Jordan) that periodically polling the file system by a LightWeightExport implementation to support notifications is acceptable.

Limitations. The LightWeightExport implementation does not permit establishing links outside of the directory structure that is being exported.
5.2.2.2 FS-Proxy

Figure 50. The FS-Proxy implements the RNS and ByteIO interfaces over other mechanisms to access files such as SMB/CIFS or ssh.

Having the ability to export any local file system into an entity that is visible on the grid namespace is a powerful but restricting option, as it is predicated on the user running a container for this. As an alternative, a user can export any hierarchical file system (even those that are not local to his machine like a Network File System) onto the grid namespace. This can be made visible via an RNS for which the grid-user has credentials to modify. The exported file system is made available as a subdirectory under this RNS. The user also needs to delegate the responsibility for exposing the regular storage/retrieval web services to some existing container. This container may/may not be local to where the user's client program is running. The functionality is, however, predicated on the fact that the container that is delegated this responsibility must have access to the shared file-system under consideration. This access needs to be provided by establishing ssh/smb/NFS/CIFS channels between the container and the file system. To reiterate, the advantage of the feature is that the container services can be delegated (hence the user need not run a container specifically) and the shared file-system is expected to support only common security measures and file-transfer capabilities without needing special installations.
5.2.2.3 RandomByteIOPortType

![Diagram of RandomByteIOPortType]

Figure 51. RandomByteI combines the ByteI interfaces and GeniiBASE. This gives us the ability to both set and get access control lists, but to get and set arbitrary resource properties (metadata).

The RandomByteIOPortType implements the RandomByteI and StreamableByteI interfaces, as well as the GeniiBase interface. When instantiated via the VCGRCGCreate() operation, instances of RandomByteIOPortType store their state — i.e., the file they represent — in the Genesis II container state directory on an underlying file system. The file system may be a network file system such as NFS or Luster, a cloud storage system such as Amazon S3, or a directly attached file system on a hard disk, flash disk, or other random access media.

Thus, Genesis II containers can act as storage servers with different performance, availability, locality, and cost characteristics. For example, we could configure a Genesis II container on a host with terabytes of cheap, slow, and relatively unreliable commodity storage. This container might be used to make local replicas of all files staged to a site with the intent of only copying a given input file once to a site. Reading from a local disk, even a slow disk, is not only usually faster than a remote read, but it consumes less-expensive wide-area shared bandwidth. Alternatively, when performance is important, high-speed disks could be used.

To support both notification-based caching and replication, the RandomByteIOPortType provides the WS-Notification interfaces that allow clients to subscribe to update events. When the file is updated, the subscriber is sent a notification with the byte range modified and the version vector of the file. (Version vectors are used when the file is replicated, which is discussed in a moment.)

The subscriber then may either request the modified byte range or invalidate their cache. As an optimization, clients also may subscribe to an update topic that sends the data in the byte range, as well.

Replication. Instances of RandomByteIOPortType support replication with a eventual consistency protocol as described in [83]. The basic idea is simple. (The details are a bit more complicated and are described in the paper.) Assume a RandomByteIOPortType file F1 has already been created and exists on container C1. Assume also Resolver (section 5.2.4.1) R1 already exists. (If not, then create a new ReferenceResolver first.) To replicate F1 on a container C2, one first extracts the EPI of F1, EPR-F1, then one calls the VCGRCGCreate() operation on C2 with construction parameters specifying that the new instance should have the EPI EPI-F1, that it is replicated, and the EPR of R1. Once the new instance is added to the list of replicas in R1, marks itself in an inconsistent state, and
begins reading its contents from the other replica, the new instance then subscribes to update persistent notifications\(^6\) at each of the other instances.

When any replica of F1 is written to, it raises the update and update with data events, updating the other replicas.

**5.2.2.4 Access to Standard Storage Systems**

The SRM interface [76] is an open standard adopted by major storage systems used in the scientific communities around international Grid activities (e.g., EGI, OSG). The University of Dresden provides and maintains an SRM-client for UNICORE6 to enable the access of scientific data stored in SRM data storage systems (e.g., dCache, DPM, BESTMAN). Among the largest scientific communities to take advantage of these SRM adoptions are those working on the four experiments at WLCG. Figure 52 provides an example of how the SRM client can be used to obtain LHC data within EGI for computational intensive HPC computations within PRACE.

---

\(^6\) A persistent notification is a Genesis II container-specific topic. When raised, they are placed into a “persistent outcall database.” The container retries these calls for up to a configurable period of time (default one month) until they get through. Note that such notifications MUST BE idempotent.
limited to UNICORE 6 services and can be used for any other WS-based services. The Registry can be configured in two major ways, first, as a local service registry referring to services at one center as shown in Figure 53. Alternatively, one globally configured registry can be used to also provide information about services from other centers as shown in Figure 54.

Figure 53. UNICORE 6 Registry used with two resources at one resource provider.

Figure 54. UNICORE 6 Registry used in a global configuration.

5.2.3.2 Information schema and services

Information about the capabilities of the services needs to be offered by each individual service, such as, for instance, the BES implementations that offer this functionality as part of their GetFactoryAttributes. This can be used to obtain “local information” from this particular service. To complement this “local information” view, an information service can be used for a more “global information” view and needs to be based on an open-standard information schema. The use of a common information schema is much more vital than the use of one single information system, as the semantic information within schema instances are key to the understanding of the broader XSEDE resources ecosystem. One example is the open standard GLUE2 specification [11]. The use of such an information service is illustrated in Figure 55.
Figure 55. Use of the GLUE2 open standard for describing XSEDE resources.

The UNICORE Common Information Service (CIS) [54] provides a broader GLUE2 interface [11] and is based on the common information provider (CIP). The BES implementations provide the information service with local information, which is then summarized up to a whole Grid infrastructure. This is shown in Figure 56. The common information service is created by UNICORE developers but not directly bound to UNICORE and, as such, can be re-used with any other middleware systems.

Figure 56. CIS Architecture in Context of UNICORE 6 key elements.

5.2.4 Infrastructure Services

5.2.4.1 Resolver

Another example of interface composition with RNS is the combination of RNS with replica catalogs used for replicated services. The idea is simple, to use RNS directories to keep track of the different replicas of a named (e.g., by an EPI) Web Services endpoint. Finding the replicas reduces to a directory lookup, the same directory lookup mechanism that is used everywhere else. Combined
with a FUSE or other file system interface, this feature allows the replica space to be managed simply by using Unix directory operation, rather than yet another tool.

Genesis II, one of the many possible Web Services realizations of the XSEDE Web Services Architecture, has a number of software components that implement the interfaces above. All of these components are contained in the Genesis II server component.

![Diagram](resolver.png)

**Figure 57.** Resolvers combine the Reference and EndpointIdentifier resolvers with RNS. This allows us to easily access different replicants of a resource via the file system interface.
5.2.4.2 EnhancedRNSPortType

![Diagram of EnhancedRNSPortType]

Figure 58. The EnhancedRNSPortType provides a full-service replicated directory service with the ability to create RandomByteIO instances directly.

The `EnhancedRNSPortType` provides the RNS 1.1 and WS-Iterator interfaces as expected, as well as the CreateFile interface that allows clients to create RandomByteIOPortType instances to be created in the same container as the `EnhancedRNSPortType` instance. Thus, when a client creates a file in a directory, they know that, unless otherwise specified, the file will be in the same container as the directory.

To support both notification-based caching and replication, the `EnhancedRNSPortType` provides the WS-Notification interfaces that allow clients to subscribe to update events. When the file is updated, the subscriber is sent a notification with the RNSEntries added or deleted and the version vector of the file. (Version vectors are used when the file is replicated, which will be discussed in a moment.)

The subscriber then may either request the modified byte range or invalidate their cache. As an optimization, clients also may subscribe to an update topic that sends the data in the byte range, as well.

**Replication.** Instances of `EnhancedRNSPortType` support replication with an eventual consistency protocol as described in [83]. The basic idea is simple. (The details are a bit more complicated, and are described in the paper.) Assume an `EnhancedRNSPortType` file \( D1 \) has already been created and exists on container \( C1 \). Assume also Resolver (section 5.2.4.1) \( R1 \) already exists. (If not, then create a new Resolver first.) To replicate \( D1 \) on a container \( C2 \), one first extracts the EPI of \( D1, EPR-D1 \), then one calls the `VCGRCreate()` operation on \( C2 \) with construction parameters specifying that the new instance should have the EPI \( EPI-FD \), that it is replicated, and the EPR of \( R1 \). Once the new instance is added to the list of replicas in \( R1 \), marks itself in an inconsistent state, and begins reading its contents from the other replicas, the new instance then subscribes to update persistent notifications\(^7\) at each of the other instances.

\( ^7 \) A persistent notification is a Genesis II container-specific topic. When raised, they are placed into a "persistent outcall database." The container retries these calls for up to a configurable period of time (default one month) until they get through. Note that such notifications MUST BE idempotent.
When any replica of D1 is written to, it raises the update and update with data events, updating the other replicas.

Because one often wants to replicate all of the files in a directory as well as their sub-directories recursively, EnhancedRNSPortType supports an additional resource property to indicate that the EnhancedRNSPortType is replicated AND that new files and directories created in that directory will be replicated on the same containers as the directory itself. We call this the replication policy sticky bit. For example, suppose D1 is replicated on C1, C2, and C3. If the file F1 is created in D1, then F1 will be replicated on C1, C2, and C3, similarly if a new directory D2 is created in D1. Note that the sticky bit is inherited in D2, so if F2 is created in D2, it also will be replicated on C1, C2, and C3.

5.2.5 Identity

5.2.5.1 IdentityProvider (X509AuthN, XSEDE KerbAuthNPortType, ECP-STS)

Recall that an STS takes a set of identity tokens conveyed by the callee in the SOAP header and returns a set (possibly empty) of identity tokens (usually signed to the callee). In Genesis II the identity tokens returned are signed GAML certificates that are delegated to the callee.

There are currently three "subtypes" of IdentityProviders in Genesis II, X509AuthN, XSEDE KerbAuthNPortType, and the InCommon ECT-STS (for Enhanced Client Protocol Secure Token Service). The ECT-STS is described but NOT yet implemented.

X509AuthN instances look for a username/password identity token in the SOAP header. (Recall that all traffic flows over TLS encrypted channels.) Each X509AuthN has its own X.509 certificate and corresponding key pairs. The X509AuthN instance keeps the username and the hashed password in the local container database. To validate identity, the X509AuthN hashes the password of the identity token and compares it with the stored hashed token. If they match, a signed GAML assertion is returned.

XSEDE KerbAuthNPortType instances use a different method to authenticate.

XSEDE KerbAuthNPortType uses a Kerberos realm to authenticate users, for example, the XSEDE Kerberos realm. The XSEDE instances of the KerbAuthNPortType will reside in a Genesis II container running on a secure machine at one of the centers. (The location of the server and its backup are the responsibility of the centralized services team.) We will call this server the IDP server. KerbAuthNPortType implements XSEDE’s requirement that users authenticate using an authentication mechanism supported by the XSEDE production baseline.

The KerbAuthNPortType expects a username/password token to be conveyed in the SOAP header as per Web Service Interoperability Basic Security Profile (WSI BSP) [86]. When invoked, the KerbAuthNPortType connects to the specified Kerberos realm and requests authentication of the username/password. If the Kerberos realm authenticates the request a GAML assertion is created by the KerbAuthNPortType service. The assertion contains the distinguished name of the user, e.g., alice@xsede.org. The assertion is signed and delegated to the caller X.509 identity using the KerbAuthNPortType private key; in the current implementation the key is the private container key.
In the current KerbAuthNPortType implementation, the user password is not retained. In a future implementation, we may want to retain the Kerberos ticket to authenticate to MyProxy and store the key for subsequent use to copy files using GridFTP.

ECP-STS instances use the InCommon Identity Provider (IDP) server Enhanced Client Protocol (ECP). The basic idea is similar to the KerbAuthNPortType implementation. Rather than performing a lookup for the password in the container database as in the X509AuthN case, or making an outcall to the Teragrid Kerberos realm, the ECP-STS loads from the database the XSEDE portal ID, the name of the InCommon institution, e.g., Indiana, the URI for the institution’s InCommon ECP IDP, and the user ID at that institution, e.g., Craig.Stewart. Then, using the ECP protocol, it calls the institution's InCommon ECP IDP, passing the user ID and password. If an appropriately signed SAML certificate is returned by the InCommon IDP, the ECP-STS generates and signs an appropriate SAML/GAML certificate.

Both the X509AuthN and XSEDE KerbAuthNPortType Implementations support the RNS 1.1 interface. Links to other STS endpoints can be linked into the identity provider’s directory. The semantics are such that when a callee invokes the “get tokens,” the implementation recursively calls the “get tokens” method on all other STSs in its directory.
For example, the user “grimshaw” STS could have links to the group “UVA” STS and to the “XSEDE” group STS. When user Grimshaw authenticates to “grimshaw,” the STS would automatically authenticate to UVA and XSEDE using the “grimshaw” credential and return the set of credentials <“grimshaw”, “UVA”, “XSEDE”>. Access control to the UVA and XSEDE STS services, and hence, membership in the groups would be controlled by group owners.

5.2.6 Accounting & Allocation

5.2.6.1 Standardized Tracking of Resource Usage across XSEDE resources

Tracking resource usage across the wide variety of XSEDE resources is a challenge because of its major heterogeneous resources. In this context it is important to ensure that tracking of resource usage is supported by a standardized schema that enables semantic interoperability when comparing usage records from resource A with resource B. Also, for the overall aggregation of the resource usage, it makes sense to agree and enforce a common usage record schema like UR that is part of X-WAVE.

The UR standard was successfully used within the European DEISA HPC-driven infrastructure and also is used within the HTC-driven infrastructure HTC. In many cases, the UR is created by resource-management system-specific adapters (e.g., Torque, SGE) as shown in Figure 61. The University of Torun and ICM provide and maintain a UNICORE 6 adoption of the RUS interface, including numerous adapters for resource management systems. A manual of how to install and configure a UNICORE RUS service is available [51].
Figure 61. The RUS Service uses URs created from RMS-specific adapters.

The tracking of storage resource usage is also part of the more recent versions of the UR. SRM implementations such as dCache, DPM, and StoRM are already adopting the storage accounting record part of UR 2.0.

5.2.6.2 Existing Tooling around Accounting Standards

Using open standards for the tracking of resource usage enables the use of broader tools not necessarily bound to a specific middleware like GENESIS or UNICORE 6. One example is the LLView system that adopted a RUS client in order to illustrate the resource usage of computational resources and works with the UNICORE 6 RUS interface. LLView is able to obtain and interpret the standard UR instances and create and illustrations as shown in Figure 62.
Since the use of UR and RUS is based on open standards, LLView also can work with other adoptions of the RUS interface as shown in Figure 63. X-WAVE thus enables an easy way of illustrating the UR with an existing tool that is broadly used around the world at HPC and other centers.

Other systems available using the UR standard are billing systems such as Swedish Grid Accounting system (SGAS) and APEL. These systems are used in EGI’s national Grid initiatives. All different tools share the fact that they work on a common usage record that enables comparisons among infrastructures and resources.

5.2.7 Help Desk & Ticketing
This section is currently empty.

5.3 Common Component Interactions in X-WAVE
We break the use cases down into a number of basic operations that are performed repeatedly in different orders to meet the campus bridging use case requirements. All interactions between components use Web Services as described above. In particular, interactions take place over
https/TLS, and identity tokens are conveyed in the SOAP header. All endpoints are referenced using WS-Addressing EPRs and implement the OGSA WSRF BP and the OGSA Basic Security Profiles.

5.3.1 Directory Lookup

The first step in all interactions in X-WAVE is to authenticate and acquire identity tokens. This begins with directory lookup, an example is shown below.

As previously mentioned, the directory lookup process works like a scalable directory lookup service. Consider a situation where a file “/server/s1” is referred to any of the software components in the Access Layer (say an `ls` command). The lookup starts with determining the root namespace (1) or “/” analogous to the UNIX-style of resolving file names to inodes. The root namespace contains a port-type/service that is capable of identifying the location of the directory “server.” This could be a file located either local to the root name-space or remote to it. Either way, the response abstracts this detail and returns an EPR (End Point Reference). When the client in the access layer receives the EPR, the lookup for “file” continues, only this time the location being queried is the location that the EPR in (2) represented. This process continues until a file level resolution is obtained.

The following diagram represents the abstract lookup of any directory, as long as it provides the EnhancedRNSPortType.
5.3.2 Authenticate

Authentication in the X-WAVE involves a number of components, not all of which are Web Services. Each are described below, and then the steps are explained.

On disk security context: The Genesis II Access Layer client keeps a copy of the security context on disk so it is available to multiple applications and can be used across application runs. The security context is a serialized Java object with:

i. The client session X.509 key pair.

ii. GAML assertions delegated to the client session keys. Typically one for the user credential and one for each group of which the user is a member. GAML certificates are SAML-like, Java-serializable, signed assertions. There are methods to sign, add assertions, get assertions, and check the signature and trust chain. Java SE security classes are used to implement GAML signing, signature checking, etc.

iii. Any username/password tokens the user has chosen to use (uncommon).

Genesis II client: The Genesis II client access layer component. This is typically performing the command-line client or the GUI client. For the general use case, the user calls login, a command-line runnable. Command-line-runables can be invoked by the Genesis-II shell, and third parties can extend the shell without recompilation.

MyProxy CA: The MyProxy CA is operated by NCSA and has several interfaces, including a Java interface that we are using.

XKA-IDP: XKA-IDP stands for XSEDE Kerberos Aware IDentity Provider. The XKA-IDP is an instance of the IdentityProvider class that uses a Kerberos realm server for authentication. We use the com.sun.security.auth.module.Krb5LoginModule, part of the Java authentication and Authorization services package, JAAS.
**ECP-IDP**: ECP-IDP stands for Extended Client Protocol IDentity Provider. The ECP-IDP is an InCommon IDP server that supports the ECP protocol for authentication using the InCommon.

**Kerberos**: This component is the XSEDE Kerberos realm hosted by NCSA.

**Teragrid Central Database**: Keeps track of all XSEDE users.

### 5.3.2.1 General Case

The general case assumes that there are no special requirements; e.g., client must use an end-entity certificate from a particular CA or that the SOAP header must carry some specific fields. In other words, a WS-Trust STS IDP (§ 4.1.9) is used to authenticate and sign/delegate an identity token.

---

**Figure 66. Sequence diagram basic authentication to an X509AuthN WS-Trust STS.**

**Steps:**

1. Client Startup - execute the `grid` command. The on-disk-security context is loaded.
2. User calls login and provides their ID and their password. The ID can be an arbitrary path to an IDP (WS-Trust STS, 5.2.5.1). By default the code looks in "/users".
3. The user ID provided is looked up in the global directory, by default in /user/ID; "/" and "/user" and "/home" (RNS resources) will be centrally managed resources (Steve McNally, NICS). The client walks the tree using the pathname lookup mechanism shown in above in "Directory Lookup."
4. The client calls the IDP `requestSecurityToken()` function passing username/password and expects a signed GAML certificate back.
3.1.1 IDP uses some implementation-defined mechanism for authentication using username/password. The default behavior is to compare username and hashed password to the username and hashed password stored in the container database.

3.1.2 Assuming there is a match and the user is validated, the IDP generates the delegated to client GAML certificate. The GAML credential is a signed assertion (signed by IDP) that the holder of the private key of the session certificate {c.private} is authorized to be {ID} — that’s the essence of delegation. A set of GAML credentials is returned, each representing an identity, each of which might model a user, a group, a role; the assertion is of a relation between ID and these named identities; each Genesis-II EPR embeds an X.509 certificates, using a standard WS way of extending an EPR with non-standard features, e.g., signing. The GAML is returned to the client.

4. *login* updates the in memory and on-disk-security context.

5. Done.

5.3.2.2 XSEDE Portal ID Case

This case is similar to the general case but calls *xsedeLogin*, which adds steps to acquire the XSEDE-specific MyProxy end-entity certificate for use as the client session certificate. The following sequence diagram and explanation describe the steps used to authenticate in XSEDE.

---

**Figure 67. Sequence diagram for authentication using the xsedeLogin tool.**
Steps:
1. Client Startup - execute the grid command. The on-disk-security context is loaded.
2. User calls xsedeLogin and provides their XSEDE portal ID and their password.
2.1 Authenticate to MyProxy - xsedeLogin (the client) calls on a Java class provided by NCSA to use MyProxy, which sets up the link and transmits requests and responses.
   2.1.1 MyProxy calls Kerberos domain (the server)? Returns Yes/No. The server usually runs at NCSA.
   2.1.2 MyProxy looks up user portal ID in teragrid central database.
   2.1.3 MyProxy returns an end-entity certificate, which means additional certificates can not be "derive" further certificates but it can sign things and things can be delegated to it.
2.2 xsedeLogin calls an internal function, keystore-login, is a Genesis-II .jar, command-line runnable, which replaces the client certificate in on-disk security context.
2.3 The user ID provided is looked up in the global directory; by default in /user/portalID. "/" and "/user" and "/home" (RNS resources) will be centrally managed resources (Steve McNally, NICS). The specific EPR resolved to is the XKA-IDP implementation, also hosted on centrally managed resource. The client walks the tree using the pathname lookup mechanism shown in above in "Directory Lookup."
2.4 The client calls the XKA-IDP passing username/password and expects a signed GAML certificate back.
   2.4.1 XKA-IDP calls XSEDE Kerberos realm.
   2.4.2 Assuming the Kerberos realm authenticates, the XKA-IDP generates the delegated to client GAML certificate. The GAML credential is a signed assertion (signed by XKA-IDP) that the holder of the private key of the session certificate {c.private} is authorized to be {portalID} — that’s the essence of delegation. A set of GAML credentials is returned, each representing an identity, each of which might model a user, a group, a role; the assertion is of a relation between portalID and these named identities; each Genesis-II EPR embeds an X.509 certificates, using a standard WS way of extending an EPR with non-standard features, e.g., signing.
2.5 xsedeLogin updates the on-disk-security context.
3. Done.

5.3.2.3 InCommon Case
The InCommon implementation presumes the implementation of an ECP-STS — an enhanced client protocol-aware STS much as in the manner of the XKA-STS described above (5.2.5.1). Assuming that such an implementation exists and that an appropriate ECP-STS instance has been created that binds the InCommon identity with the XSEDE Portal ID, then authentication will follow the same steps as in the General Case (§5.3.2.1) above.
Figure 68. Sequence diagram for authentication to an InCommon WS-Trust STS.

Steps:
1. Client Startup - execute the grid command. The on-disk-security context is loaded.
2. User calls login and provides their ID and their password. The ID can be an arbitrary path to an IDP (WS-Trust STS, 4.1.9). By default the code looks in "/users".
3. The user ID provided is looked up in the global directory, by default in /user/ID; "/" and "/user" and "/home" (RNS resources) will be centrally managed resources (Steve McNally, NICS). The client walks the tree using the pathname lookup mechanism shown in above in "Directory Lookup."
   3.1 The client calls the IDP requestSecurityToken() function passing username/password and expects a signed GAML certificate back.

3.2 ECP-STS loads from the database the XSEDE portal ID, the name of the InCommon institution (e.g., Indiana), the URI for the institution's InCommon ECP IDP, and the user ID at that institution (e.g., Craig.Stewart). Then, using the ECP protocol, it calls the institution's InCommon ECP IDP, passing the user ID and password. If an appropriately signed SAML certificate is returned by the InCommon IDP, the ECP-STS generates and signs an appropriate SAML/GAML certificate.
   3.2.1 The GAML or a fault is returned to the client.
4. login updates the in memory and on-disk-security context.
5. Done.
5.3.3 File Read/Write

The first step is to authenticate as described above in 5.3.2. Once authenticated, accessing resources in the InCommon case is no different from other authentication cases. Recall the lookup process (as discussed in §5.3.1), here diagramed in reference to a named file and directory.

![Diagram](image)

Figure 69. Looking up the file /home/marconi/file1.

Figure 70 shows the lookup to retrieve the file endpoint and the read process on the specified RandomByteIO file.

![Diagram](image)

Figure 70. Reading from the file /home/marconi/file1.

The Read functionality provided to clients in the Access Layer uses the ByteI0 service exposed by the Enhanced RNS Port Types. Once the client receives the EPR of the file that is being read to, the read request is issued to the container where the ByteI0 service is running. The service responds with the relevant data.

The following diagram shows the lookup to retrieve the file endpoint and the write process on the specified RandomByteIO file.
Figure 71. Writing to the file /home/marconi/file1.

Similar to the above, once the client receives the EPR of the file that is to be written to, the write request is issued to the container where the ByteIO service is running. The service responds with a status code indicating the success or failure of the request.

5.3.4 File Create

Consider creating a file “/home/user/file1.” The directory lookup mentioned in §5.3.1 can be used to obtain the EPR of the resource where the file is to be created (in this case, user). This EPR represents a resource that implements the Enhanced RNS. Once this RNS is obtained, the file can be
created by invoking the create_file method exposed by the port type, another aspect of file creation is to ensure that the artifact that implements the EnhancedRNS (user) is made aware of the location where the file was created.

5.3.5 File Delete

![UML Diagram]

Figure 73. Deleting a file.

Consider removing a file “/home/user/foo.txt.” The directory lookup mentioned in the previous section can be used to obtain the EPR of the resource to be removed (in this case, foo.txt). This EPR represents a resource that either can be an EnhancedRNS or a ByteIO port type depending on whether the resource represents a directory or file respectively. Once this EPR is obtained, the file can be removed by invoking the remove method exposed by either of these port types. Another aspect of file/directory removal is to ensure the file’s parent artifact that implements the EnhancedRNS (user) is made aware of the file removal. This can be done using the remove method.
5.3.6 Create Export

Creating an export involves having a container running at the point at which the filesystem to be exported resides. When a directory is exported into a resource namespace, a VCGR Create is issued to the lightweightExport service exposed by the container that resides local to the filesystem being exported. The parameters are the local file system path on the server to export and whether it should be read-only. This returns an EPR for the directory. In the future, when the container receives lookup calls for this resource, the call is proxied onto the local filesystem. At this point, the client uses the directory lookup service as previously explained and obtains the EPR of the Enhanced RNS Port Type that is the destination of the export. Analogous to File Create, an add call to this resource maps the EPR returned by the Export Service to the destination.
5.3.7 Write to a replicated file (Tesla)

Figure 75. Writing to a replicated file.

The consistency protocol enforced is known as eventual consistency. The protocol and implementation are described more fully in [83]. Any replica of a file may be written to. Each replica is subscribed to updates on all other replicas. When an update arrives, it is written to disk and the update event is published using the persistent notification method of the Genesis II containers. A persistent notification is placed in a database of pending notifications and the notification is repeated until there is a positive acknowledgment that the notification was received or a bounded specified period has passed. The persistent notification database survives crashes and re-boots. The notification contains the data written and the version vector of the file that first received the update. All of this takes place in a database transaction.

5.3.8 Write to a replicated directory (Bell)

Figure 76. Writing to a replicated directory.

The protocol is similar to the writing to a replicated file protocol described above. The difference is that when writing to a file, "last writer wins" is the usual accepted semantics. When two or more clients are writing to the same directory — in particular, creating new files with the same name — "last writer wins" can generate garbage file resources and cause confusion to users. To address this
problem, the protocol has been constructed to guarantee that no files are lost when multiple concurrent clients create the same file name in replicas before the updates can propagate. The algorithm is described fully in [83].

5.3.9 Run a job directly on a BES

As an alternative to submitting a job onto a queue and letting the queue follow its scheduling policy, the run command can be used to submit a job onto a BES directly. Once a job is submitted in the form of an Activity, the BES Factory PortType (shown below as a GeniiBESPortType) returns an EPR for that job that can be used to monitor/change its properties over its lifetime. This EPR is persisted in the client to support crash recovery in the client. Once the job is submitted, the files/data needed by the job are read directly using the ByteIO service that exposes the file. This process is completely decoupled from the client. Although not shown in this diagram, job lifecycle can be managed even without the polling scheme. A pub-sub notification system can help the client realize job status updates dynamically.

Figure 77. Executing a job directly on a BES. The job includes file staging using the GFFS.
5.3.10 Qsub a job on a grid queue (GQ)

Figure 78. Executing a job via a grid queue. The client submits the JSDL activity to the grid queue. The grid queue saves the activities to disk and subsequently runs and monitors the jobs on BES Factory PortType instances.
5.3.11  Execute a DAGMAN workflow on workflow engine

```plaintext
sd  execute a DAGMAN workflow on JLO:workflow engine

client : UserInteface
JLO : WorkflowEngine
GQ : QueuePortType

ref
lookup

1: subWorkflow()
2: string

JLO:WorkflowEngine parses workflow and stored in local database

3: submitJobs()

job can run at any later time
```

Figure 79. DAGMAN Workflow Engine. The workflow engine (JLO) receives a DAGMAN workflow graph from the client. It then parses the workflow, stores the set of sub-tasks into its local database, and starts tasks whose preconditions are met via the grid-queue/metascheduler. It then monitors progress of the jobs. As jobs complete, it checks whether new jobs can now run and, if so, starts them.

5.3.12  Quality Attribute - Availability

Files and directories can be replicated as described in [83]. Suppose a client attempts to read from a replicated directory (/marconi) but the directory resource instance is unavailable; for example, the GIU on which the instance is located is unreachable or has crashed. What will happen is that the read attempt will fault. At that point the client library examines the EPR of the directory resource to determine if it implements the WS-Naming profile and has an embedded resolver EPR. If it does, the client calls the resolver method passing in the unique identifier (EPI) of the directory and rebinds the EPR. The resolver returns the address (EPR) of another instance, and the client proceeds as above. This protocol is described in both the WS-Naming specification and in [41].
Figure 80. In this scenario the client reads from an instance of the file "marconi." The instance is down, so the client experiences a fault. The client re-binds the EPR using the resolver embedded within the EPR for "marconi" and is redirected to a replica. The read then proceeds normally.
5.3.13 Quality Attribute – Performance

One optimization made to the distributed file system that the GFFS aims to be is the parallelization of read tasks. Reads on the ByteIO service that expose a particular file tend to be asymptotically limited by the size of the buffer on the receiving end. Although most operating systems provide options to change the size of the buffer available to the TCP stream, these require root privileges. To overcome this issue, parallel streams of TCP are opened, and the file is read in a parallel fashion and assembled at the client. At this point, there is no optimization on write since there are counter-intuitive performance-related problems that arise at the server when trying to guarantee an atomic file-write.

A second approach is to have the installer of the container where the ByteIO services and the client are located change the TCP buffer sizes in the operating system. This is easily accomplished in Linux using the /proc file system, if one has root. See for example: http://www.cyberciti.biz/faq/linux-tcp-tuning/.

5.4 The Package View

Two software configuration items produced by the XSEDE Software Development & Integration team make up the X-WAVE, the Execution Management Package (EMS) and the Global Federated File System package (GFFS). When deployed and configured properly, these two configuration items implement the XSEDE Wide Area Virtual Environment (X-WAVE).

The Execution Management Services CI is concerned with specifying, executing, and more generally, managing jobs in the XSEDE grid. EMS capabilities include, but are not limited to:
• The ability to specify both single jobs and parameter space jobs in JSDL. Specified jobs may be sequential jobs or parallel (MPI) jobs.

• The ability to manage JSDL jobs through their lifetime, from specification, submission to a compute resource, status checking and management during execution, to final cleanup.

• A grid queue (metascheduler) that matches jobs to a defined, configurable set of execution services and load balances between them.

• The ability to specify either a single compute resource as a target (e.g., a particular queue on Ranger) or to specify a global metascheduler/queue as the target and have the metascheduler select the execution endpoint.

• The ability to add compute resources (e.g., queues on specific machines such as Ranger, Alamo, Kracken, or local campus queues such as Centurion at UVA) into the XSEDE namespace and subsequently target JSDL jobs at them.

• The ability to create metaschedulers/queues and configure them to use (schedule on) different compute resources.

• A command-line interface (CLI) to interact with grid compute resources.

• A graphical user interface (GUI) to interact and manage the backend grid compute resources. This includes, but is not limited to, tools to create and execute JSDL job descriptions, manage grid queues, manage access control to resources, etc.

• A set of Java classes (and associated APIs) to interact and manage the backend grid resources.

The Global Federated File System presents a filesystem-like view of diverse resources’ types located at service providers, campuses, research labs, and other institutions. Resources (e.g., files, directories, job submission queues) are mapped into a single global path-based namespace. Resources can be accessed by their path name in a location, replication, migration, and failure-transparent manner. Resources can be accessed via command-line tools (a grid shell), a graphical user interface, or via the user’s local file system and a FUSE mount. The GFFS provides a number of capabilities. These capabilities include, but are not limited to:

• A single, secure, shared (S3) global namespace for a diversity of resource types; for example, files, directories, execution services, execution queues, secure token services, and executing jobs.

• A three-level naming scheme consisting of location-independent human-readable names (paths) that map globally unique resources’ identities that in turn can be mapped (bound) to one or more resource instances. Collectively the three layers provide an easy-to-use namespace that transparently handles heterogeneous configurations for location, failure, replication, migration, and implementation.

• Securely map (share) Service Provider (SP), local, lab, and campus data into the shared global namespace.

• Securely map (share) SP, local, lab, and campus compute resources into the global namespace.

• Securely map (share) SP, local, lab, and campus identity resources into the global namespace.
• Transparent access from both campuses and centers to the global shared namespace via either the filesystem (e.g., FUSE) or via command-line tools and libraries. Such access includes the ability to perform create, read, update, and delete operations on file, directory, and other resource types.

• A command-line interface (CLI) to interact with backend grid resources, in particular, to Open Grid Forum RNS, ByteIO, WS-Naming, and BES services, as well as WC3 WS-Trust Secure Token Services.

• A graphical user interface (GUI) to interact with and manage the backend grid resources.

• A set of Java classes (and associated APIs) to interact with and manage the backend grid resources.

• Integration with existing, legacy XSEDE Kerberos, and MyProxy authentication mechanisms.

The above capabilities are delivered via a set of distinct packages (or sub-configuration items). These are the UNICORE 6 client and server packages and the Genesis II client and server packages. The UNICORE 6 and Genesis II client packages implement the X-WAVE access layer, and the UNICORE 6 and Genesis II servers implement much of the services layer of X-WAVE.

5.4.1 UNICORE 6 Server Package

The UNICORE 6 server package includes the UNICORE 6 service container and a SOAP engine that is capable of sending and receiving SOAP-based WS messages’ exchanges. The packages of different standard implementations are well encapsulated from the core container.

Figure 82. UNICORE 6 server package with standards, OGSA BES and ByteIO.
5.4.2 UNICORE 6 client package
UCC, URC.

5.4.3 Genesis II server container package
The Genesis II server package executes in an Apache Axis Web Services Engine in a Java Virtual Machine. The package is delivered as a set of platform-specific installers, e.g., 64-bit Linux, Windows 7, MacOS, etc. Each installer contains all of the files needed to run the server, a Java virtual machine, all of the platform-specific JNDI files, the JAR files, scripts, and the deployment configuration. The installers are self-extracting programs that unpack the contents and interact with the user to localize the deployment options.

5.4.3.1 Genesis II Grid Container
Genesis II is implemented using Web Services to perform the tasks of the various system components — such as components to implement file and directory resources, execution service resources, grid queue/metascheduler, user identity provider resources, etc. Each of these components implements a well-defined set of functionality with interfaces described by WSDL documents.

![Diagram of Genesis2ServiceContainer](image)

**Figure 83.** The Genesis2ServiceContainer hosts all of the Genesis II service components. All actions taken by a Genesis2ServiceContainer are performed with the user ID of the person who installs the container. There is no need for root privilege.

Under the hood, Genesis II uses Apache/Axis to host these Web Service components, and each application server deploys the code for all Genesis II service types (each of which is named using a final porttype that is, in turn, mapped to the set of WSDL Port types implemented). We call a
Genesis II Web application server a Genesis II grid container ("grid container" for short), as it contains Genesis II services and service instances.

The following components (described earlier) are packaged in a grid container:

- RandomByteIOPortType
- EnhancedRNSPortType
- GeniiBESPortType
- BESActivityPortType
- QueuePortType
- VCGRContainerPortType
- X509AuthNPortType
- KerbAuthNPortType
- FS-Proxy
- LightWeightExport
- Resolver

Most Genesis II services can create instances of their type, which we call Genesis II grid resources ("resources" for short). For example, the service RandomByteIOPortType can create grid resources that adhere to the standard OGSA ByteIO interface, i.e., resources that embody file functionality. Once created, each resource is a unique, individual entity with its own state, distinct from other entities — even those with the same interface. Therefore, multiple unique resources can be created from the same service on the same Genesis II grid container. For example, many RandomByteIO file resources can be created on the same grid container, each with a different state. In this case, the different state includes the different file contents. Similarly a single grid container can house a number of grid queues, a number of execution services, a number of RNS directories, etc.

All grid resources support the OGSA WSRF Basic Profile and OGSA WSRF Basic Security Profile. Thus all grid resources have a resource properties document and a mechanism to get resource properties. Resource properties typically consist of data stored in a resource properties database. This database is “owned” by the container and is accessible only by the userid of the container.

5.4.3.2 Authorization

Like many web services containers, Apache Axis allows programmers to insert pre- and post-processing functions wrapped around service invocations. In the Genesis II container, one of these service wrappers is a pluggable authorization module that determines whether the service invocation will proceed or whether a security fault will be thrown. (The point of call is the Policy Enforcement Point and the module itself is the Policy Decision Point in IETF 3198 terms.)

The current default authorization module implements access control lists. Each resource has an access control list for read, write, and execute permission. Each operation in an interface is mapped to be either an R, W, X, or A, and the R, W, and X correspond to their Unix meanings. The A indicates that all clients can call this operation — it is not subject to access control. The access control lists can be manipulated using either the command-line tool with Unix-like syntax, chmod myfile +r /users/grimshaw, or using the ACL manager tools in the Access Layer Genesis II GUI. (See below.)

Access control lists are stored as resource properties associated with each grid resource in the container databases.

When a Web Service invocation arrives at a Genesis II container, the following occurs:

1. TLS session is established between client and server.
a. Client authenticates server and verifies that the server certificate is trusted; i.e., there exists a trust chain from the server certificate to a certificate authority in the client's trust store.
b. Symmetric session key established.
c. Server authenticates client certificate as above. Server stores validated client X.509 certificate in local session state for later use.

2. Axis calls Genesis II security handler —

```java
public void checkAccess(
    Collection<GIICredential> authenticatedCallerCredentials,
    IResource resource, Class<?> serviceClass, Method operation)
throws PermissionDeniedException, AuthZSecurityException, ResourceException
```

a. Handler extracts and validates calling security context and credentials.
b. Handler checks credential against container admin certificate. Each container may be configured with a certificate that allows bypass of the access control lists. If the client possesses the container admin certificate, then the handler returns with an "ok". Else,
c. Handler looks up access control module to use.
   i. Handler looks up file resource access control list.
   ii. Handler compares access control list to calling context credentials and type (R, W, X) of the call.
       If permitted, handler returns, else a security fault is thrown.

5.4.4 Genesis II Access Layer (client) Package
The Genesis II client package is a self-extracting installer that interacts with the user to fine-tune deployment on a client host filesystem. The installer contains a Java JVM, the deployment configuration files, and all necessary JAR and JNDI files. Clients are currently available for Windows, 32- and 64-bit Linux, and MacOS.

The client software can be deployed on a shared filesystem and used by many different users and hosts (e.g., if deployed on an NFS file system.) Each user must have his or her own session state directory, referred to by $GENII_USER_DIR in Linux and MacOS.

The client is initiated using the "grid" command, which places the user in the Genesis II command-line client described earlier. The "grid" command is used for all Genesis II Access Layer functionality, command-line interface, graphical user interface, and FUSE file system driver.

5.4.4.1 Client Session Certificates
In Genesis II, client sessions (e.g., the command-line client, the graphical user interface, or the FUSE file system driver) have an X.509 session certificate. This certificate is stored in the directory referred to by $GENII_USER_DIR. Unless another certificate is specified, this will be a self-signed end-entity certificate used for SSL connections and for signing documents and certificates. This certificate by itself is useless and has no privilege. It can be used only to sign documents, proving that the session is the holder of the corresponding private key.

---

8 The implementation uses the Java SE security classes. SSL sessions are cached whenever possible for performance reasons.
The client also can specify a credential (with both public and private key) to use, typically from local storage.

When the client uses the xsedeLogin tool, the myproxy services is used to acquire a certificate that will be used for the session; thus the client session certificate is an XSEDE-generated X.509 certificate.

5.4.4.2 Client credential "wallet"
The client maintains a list of credentials — including delegated credentials — that is included in the SOAP header in all outcalls. GAML certificates (5.1.8.6) are carried in the calling context element of the SOAP header. Other credentials (e.g., a username/password element) are passed in the SOAP header as specified in the WSI Basic Security Profile. The current set of credentials is displayed using the whoami tool as shown below. (Note: verbosity is LOW; the full certificates can be shown with verbosity set to HIGH.)

```
XCG:\$> whoami
Client Tool Identity:
  (CONNECTION) "Andrew Grimshaw"

Additional Credentials:
  (USER) "grimshaw" -> "Andrew Grimshaw"
  (GROUP) "uva-idp-group" -> "Andrew Grimshaw"
  (GROUP) "xsede-admin-group" -> "Andrew Grimshaw"
  (GROUP) "xsede-test-group" -> "Andrew Grimshaw"
```

Figure 84. A sample screenshot from the Genesis II command-line client showing the credential wallet for user “grimshaw.”

In this example the user, grimshaw, is authenticated using his XSEDE portal ID and is using a myproxy certificate as his session (client) certificate. He has four GAML certificates that are delegated to his client session certificate: his user "grimshaw" certificate, and the "uva-idp-group," "xsede-admin-group," and "xsede-test-group" certificates.

5.4.5 UNICORE 6 Access Layer Package
Details on this component will be provided in a future version of this document.

6 XUAS
We now present the XUAS components of the XSEDE architecture. We describe the services layer interfaces (§6.1), the services layer component types that are defined in terms of those interfaces (§6.2), and the packages that are used in the current implementation to deploy those components (§8).

6.1 XUAS Services Layer Interfaces
We describe here the interfaces that are implemented by the various XUAS components. See §6.2 for a description of how interfaces map to components.
6.1.1 XUAS Execution Management Service Interfaces

XUAS defines two execution management service interfaces, a REST rendering of the OGF-defined BES/JSDL interface and the XUAS Compute REST interface.

6.1.1.1 REST-based BES/JSDL interface

*Description:* Interface for submitting, monitoring, and managing a job to a remote compute resource, using a REST binding to BES and JSDL.

*Specification:*

- It would be straightforward to define and implement a REST binding for BES and JSDL that would be compliant with the Web/Cloud Architecture. However, such a binding has not yet been defined or implemented.

6.1.1.2 XUAS Compute REST interface

*Description:* Higher-level interface (i.e., higher level than REST-based BES/JSDL) for defining, submitting, and managing individual tasks and many-task (high-throughput) workflows.

*Specification:*

- This is a future component that is not yet defined.

6.1.2 XUAS Data Management Service Interfaces

XUAS defines four data management service interfaces: GridFTP relay, XUAS Transfer REST, XUAS Storage REST, and XUAS CLI. (The related GridFTP service interface is defined in §7.1.1.1.)

6.1.2.1 GridFTP relay interface

*Description:* This gsssh-based interface is for tunneling GridFTP control channels so that a GridFTP server can be behind a NAT or firewall that permits only outbound connections.

*Approach:*

- Client running behind a firewall or NAT connects to GridFTP relay and establishes a tunnel.
- Client gets back a relayhost:relayport from the relay service.
- Client connects that tunnel to a local GridFTP server.
- GridFTP clients use relayhost:relayport to connect to the GridFTP server via the relay tunnel

*Specification:*

- Details on this component will be provided in a future version of this document. Globus Transfer has something today, but we may want to generalize it so it can be used for clients other than Globus Transfer.

6.1.2.2 XUAS Transfer REST interface

*Description:* File and folder transfer, synchronization, and management

*Overview:*

- Type definitions:
  - **task-id** (type=Globally Unique IDentifier/GUID): Uniquely identifies a transfer request.
- **endpoint** (type=string): A human-friendly name for a storage system to/from which data can be transferred by XUAS Transfer. The endpoint definition includes one or more GridFTP server addresses, and various other information.
  - The format of an endpoint is always `<account-name>#<endpoint-name>`.
- **path** (type=string): Unix-style file path.
- **path-expression** (type=string): A path with wildcards.
- **field** (type=string): A named attribute/value pair returned from various commands.

- **Operations** (Note: These are currently defined in terms of the CLI operations but instead should be changed to the REST resources. But I’m not sure how that will work with UML.):
  - **transfer-start** source-endpoint:source-path destination-endpoint:destination-path [options]
    - Submit a transfer request.
    - Returns task-id for this request.
  - **transfer-status** [task-id] [options]
    - Returns fields with summary information about a set of transfer requests.
    - Options allow selection of which requests (e.g., all, running, specific task-id, etc), and which fields to return.
  - **transfer-details** task-id [options]
    - Returns fields with detailed information about the status of a particular transfer request.
  - **transfer-events** task-id [options]
    - Returns detailed events about a particular transfer request.
    - Events include start and completion of actions, faults encountered, etc.
  - **transfer-wait** task-id [options]
    - Wait for transfer request to complete.
  - **transfer-modify** task-id [options]
    - Modify an existing request, including deadline and label.
  - **transfer-cancel** task-id [options]
    - Cancel a transfer request.
  - **transfer-delete** endpoint:path-expression [options]
    - Delete (optionally recursively) files and directories matching the path-expression.
  - **ls** endpoint:path-expression [options]
    - List files matching the path-expression.
  - **mkdir** endpoint:path [options]
- Create a new directory.
  ○ **endpoint-add** endpoint [options]
    - Create a new endpoint definition.
  ○ **endpoint-modify** endpoint [options]
    - Modify an endpoint definition.
  ○ **endpoint-list** [options]
    - List endpoints.
  ○ **endpoint-rename** endpoint-oldname endpoint-newname [options]
    - Rename and endpoint.
  ○ **endpoint-remove** endpoint [options]
    - Delete an endpoint.
  ○ **endpoint-activate** endpoint [options]
    - Delegate a security credential to the service that it can use for the user’s interactions with the specified endpoint.
  ○ **endpoint-deactivate** endpoint [options]
    - Remove the security credential the service has for the user’s interactions with the specified endpoint.

*Specification:* See https://transfer.api.globusonline.org/.

### 6.1.2.3 XUAS Storage REST interface

**Description:** Cloud storage service

**Overview:**

- Extends: XUAS Transfer REST Interface (§6.1.2.2).
- Type definitions:
  ○ **volume** (type=string): Name of storage volume (equivalent to XUAS Transfer endpoint).
  ○ Others from XUAS Transfer.
- Operations (Note: These are currently defined in terms of the CLI operations, but instead should be changed to the REST resources. But I’m not sure how that will work with UML):
  ○ **storage-create** volume [options]
    - Create new storage volume.
    - Options for storage location(s), size, replication policy, retention policy, etc.
  ○ **storage-share** volume:path [options]
    - Share (part of) a volume with users and/or groups.
  ○ **storage-delete** volume [options]
- Delete a storage volume and its contents.
  - **storage-snapshot** `volume:path newvolume [options]
    - Create a new volume that is a point-in-time consistent snapshot of (part of) an existing volume.
  - **storage-version-revert** `volume:path time [options]
    - Revert a file to last version before specified time.
  - **get** URL
    - Read file contents via normal HTTP GET.
  - **put** URL
    - Replace file contents via normal HTTP PUT.

- All XUAS Transfer REST interfaces

**Specification:**
- Details on this component will be provided in a future version of this document.

### 6.1.2.4 XUAS command-line interface

**Description:** (gsi)ssh-accessible restricted shell that provides command-line access to equivalent functionality found in the various XUAS REST interfaces.

**Specification**
- CLI command reference [82].
- `ssh <user>@cli.globusonline.org help`.

### 6.1.3 XUAS Discovery and Information Service Interfaces

XUAS defines two discovery and information service interfaces, integrated information service and XUAS Nexus REST.

#### 6.1.3.1 Integrated Information Service

**Description:** Simple REST & XML catalog service for discovering information about a modest number of slowly changing items.

**Specification:**
- Details on this component will be provided in a future version of this document. This is a simple XML document repository with a REST interface.

#### 6.1.3.2 XUAS Nexus REST interface

**Description:** Scalable resource catalog service.

**Specification:**
- Details on this component will be provided in a future version of this document. This is future work. A detailed design has not been created yet, but it will be an extension of the current Nexus interface.
6.1.4 XUAS Identity Service Interfaces
Identity service interfaces are concerned with authentication and authorization. XUAS defines four such interfaces: OAuth 2.0, X.509 credential retrieval, XUAS Nexus REST, and XUAS Nexus LDAP.

6.1.4.1 OAuth 2.0 interface
*Description:* OAuth 2.0-compliant authentication and authorization granting interface.

*Specification:* OAuth 2.0 [6].

6.1.4.2 X.509 credential retrieval interface
*Description:* REST interface for retrieving a short-term X.509 user certificate based on an OAuth 2.0 access token.

*Specification:* MyProxy interface for REST credential retrieval using OAuth access token [15].

6.1.4.3 SAML authentication interface
*Description:* Details on this component will be provided in a future version of this document.

*Specification:* Details on this component will be provided in a future version of this document.

6.1.4.4 XUAS Nexus REST interface
*Description:* REST interface to federated identity, group, and profile management.

*Specifications:*

- Globus Nexus API Documentation [3].
- OAuth 2.0 [6].

6.1.4.5 XUAS Nexus LDAP interface

*Overview:* Information will be provided in a future version of this document.

*Specifications:*

- Details on this component will be provided in a future version of this document. The material to be provided will specify how Nexus groups are represented in LDAP.

6.1.5 XUAS Accounting and Allocation Service Interfaces
Information on these interfaces will be provided in a future version of this document.

6.1.6 XUAS Help Desk and Ticketing Service Interfaces
Details on this component will be provided in a future version of this document. We probably need to be thinking in terms of a REST-based ticket interchange interface that can be layered on XSEDE help desk, Zendesk, campus helpdesks, and other systems.

6.2 XUAS Services Layer Component Types
We organize the Web/Cloud service-layer component types in three groups: execution management, data management, and identity.
6.2.1  XUAS Execution Management Component Types
We define two execution management component types: the REST-based BES/JSDL component — which is intended to be used to provide access to a resource-layer compute resource — and the Globus Compute component.

6.2.1.1  REST-based BES/JSDL component
Description: Details on this component will be provided in a future version of this document.
Provides: REST-based BES/JSDL interface (§6.1.1.1).
Uses: Resource layer compute resources.
Deployment view: One per compute resource.

6.2.1.2  Globus Compute component
Description: Details on this component will be provided in a future version of this document.
Provides:
• Globus Compute thin-client Web GUI (§3.1.6).
• XUAS Compute interface (§6.1.1.2).
Uses:
• REST-based BES/JSDL interface (§6.1.1.1).
• OAuth 2.0 interface (§6.1.4.1).
• X.509 credential retrieval interface (§6.1.4.2).
Deployment view: One global instance, operated by the University of Chicago.

6.2.2  XUAS Data Management Component Types
We describe three data management component types: Globus GridFTP relay, Globus Transfer, and Globus Storage. The related Globus GridFTP component is described in §7.2.1.1.

6.2.2.1  Globus GridFTP relay component
Description: Details on this component will be provided in a future version of this document.
Provides:
• GridFTP relay interface (§6.1.2.1).
• GridFTP interface (§7.1.1.1; control channel only).
Uses: None.
Deployment view: One global instance, operated by the University of Chicago.

6.2.2.2  Globus Transfer component
Description: Details on this component will be provided in a future version of this document.
Provides:
• Globus Transfer thin-client Web GUI (§3.1.4).
• XUAS Transfer REST interface (§6.1.2.2).
Uses:

- GridFTP interface (§7.1.1.1).
- OAuth 2.0 interface (§6.1.4.1).
- X.509 credential retrieval interface (§6.1.4.2).

Deployment view: One global instance, operated by the University of Chicago.

### 6.2.2.3 Globus Storage component

**Description:** Details on this component will be provided in a future version of this document.

**Provides:**

- Globus Storage thin-client Web GUI (§3.1.5).
- XUAS Storage REST interface (§6.1.2.3).
- XUAS Transfer REST interface (§6.1.2.2).

**Uses:**

- GridFTP interface (§7.1.1.1).
- Amazon S3 HTTP object store interface [1].
- OAuth 2.0 interface (§6.1.4.1).
- X.509 credential retrieval interface (§6.1.4.2).

Deployment view: One global instance, operated by the University of Chicago.

### 6.2.3 XUAS Identity Component Types

We describe three identity component types: CILogon, MyProxy OAuth, and Globus Nexus.

#### 6.2.3.1 CILogon component

**Description:** Allows a service to authenticate a user based on his or her InCommon credentials, and returns an X.509 certificate for that user.

**Provides:**

- OAuth 2.0 interface (§6.1.4.1).
- X.509 credential retrieval interface (6.1.4.2).

**Uses:**

- SAML authentication interface (§6.1.4.3).
- OpenID interface [34].

Deployment view: One global instance, operated by NCSA.

#### 6.2.3.2 MyProxy OAuth component

**Description:** Allows authentication to a local identity domain via OAuth 2.0, and returns a short-term user certificate tied to that local identity.

**Provides:**

- OAuth 2.0 interface (§6.1.4.1).
• X.509 credential retrieval interface (6.1.4.2).

Uses: Resource layer identity service.

Deployment view: One per identity domain (e.g., on a campus).

6.2.3.3 Globus Nexus component

Description: Details on this component will be provided in a future version of this document.

Provides:

• Globus Nexus thin-client Web GUI (§3.1.3).
• XUAS Nexus REST interface (§6.1.3.2).
• OAuth 2.0 interface (§6.1.4.1).
• SAML authentication interface (§6.1.4.3).
• XUAS Nexus LDAP interface (§6.1.4.5).

Uses:

• OAuth 2.0 interface (§6.1.4.1).
• X.509 credential retrieval interface (§6.1.4.2).
• OpenID interface [34].

Deployment view: One global instance, operated by the University of Chicago.

6.3 XUAS Packages

We describe two sets of packages, the Globus Connect packages and the Globus Connect Multi User packages. Each is provided for a range of operating systems (OS’s).

6.3.1 Globus Connect package

Description: This package is designed for users who want to enable access to a storage resource to which they have access, without requiring admin support. The storage resource could be, for example, desktop, laptop, or a server not otherwise running GridFTP. Some specifics:

• Intended for installation by end user.
• Globus GridFTP component, packaged for easy installation on Windows, Mac, and Linux.
• Does not require admin privileges to install.
• Only works for the one user against that storage system.
• Works behind NAT/firewall by only doing outbound connections.
• Creates a Globus Transfer endpoint.

Combines:

• Globus GridFTP component (§7.2.1.1).
• Globus GridFTP relay client component (§6.2.2.1).
• Configuration and monitoring (not client) GUI component.
• Installation glue.

Deployment view:
• One per storage resource.
• Other alternatives:
  ○ Globus GridFTP package (§7.3.1).
  ○ Globus Connect Multi-User package (§6.3.2).

6.3.2 Globus Connect Multi User package
Description: This packaging makes it easy for a systems administrator to enable access to a storage resource for multiple users (e.g., a campus cluster or departmental server).

Combines:
• Globus GridFTP component (§7.2.1.1).
• MyProxy OAuth component (§6.2.3.2).
• Installation glue.

Deployment view:
• One per storage resource.
• Other alternatives:
  ○ Globus GridFTP package (§7.3.1).
  ○ Globus Connect package (§6.3.1).

7 XSEDE Architecture Common Elements
Several services layer interfaces, component types, and packages are common to both X-WAVE and XUAS. We describe the common interfaces (§7.1), the common component types (defined in terms of those interfaces) (§7.2), and the common packages that are used in the current implementation to deploy those common components (§7.3).

7.1 Services Layer Interfaces (Common)
We describe here the common services layer interfaces that are implemented by the various common components. See §7.2 for a description of how interfaces map to components.

7.1.1 Data Management Service Interfaces (Common)
The XSEDE Architecture currently defines one common data management service interface, namely GridFTP.

7.1.1.1 GridFTP interface
Description: The GridFTP protocol extends the FTP protocol with features designed to enable secure, reliable, high-performance execution.

Specifications:
• IETF RFC 959: File Transfer Protocol (FTP), J. Postel, R. Reynolds (October 1985) [70].
• IETF RFC 2228, FTP Security Extensions, M. Horowitz and S. Lunt (October 1997) [45].
• IETF RFC 2389, Feature negotiation mechanism for the File Transfer Protocol, P. Hethmon and R. Elz (August 1998) [44].
• IETF RFC 3659, Extensions to FTP, P. Hethmon (March 2007) [43].
• OGF GFD-R-P.020: GridFTP: Protocol Extensions to FTP for the Grid [9].

7.2 Services Layer Component Types (Common)

We describe here the common services layer component types that implement the interfaces of §7.1. See §7.3 for a description of how components are implemented by packages.

7.2.1 Data management components (Common)

We define one common data management component, namely Globus GridFTP.

7.2.1.1 Globus GridFTP component

Description: Globus GridFTP [8] provides an implementation of the GridFTP protocol (§7.1.1.1). It implements a variety of features designed for high-speed data movement, such as striping over multiple data movers, parallel streams, and pipelining for small files [22]. A pluggable architecture allows it to target different storage systems, including POSIX file systems, HPSS, and HDFS.

Provides:
• GridFTP interface (§7.1.1.1).

Uses:
• Resource layer storage resources.

7.3 Packages (Common)

We describe common data management packages.

7.3.1 Globus GridFTP native packages

Description: Packaging of the Globus GridFTP component using OS-native packaging and configuration systems. This packaging is designed for sophisticated systems administrators, and for complex deployments of the Globus GridFTP component that may need specialized configuration.

In more detail, the GT 5.2.0 release provides source and binary RPM packages for CentOS 5, Fedora 13, 14 and 15, RedHat 5, and Scientific Linux 5.5 and 6.1, and a set of .deb packages for several Debian and Ubuntu versions, including Debian 5.0 "lenny" and 6.0 "squeeze," and Ubuntu 10.10 and 11.4.

Provides:
• GridFTP interface (§7.1.1.1).

Uses:
• Resource layer storage resources such as POSIX-compatible file systems, High-Performance File System (HPSS) [85], and Hadoop Distributed File System (HDFS) [19].

Deployment view:
• One per storage resource. This packaging is designed for sophisticated systems administrators and complex deployments that may need specialized configuration.

• Other alternatives that provide for simpler deployment:
  ○ Globus Connect package (§6.3.1).
  ○ Globus Connect Multi-User package (§6.3.2).

8 Deployment

This section is still in draft form. It needs to be revised, in particular, to incorporate XUAS elements.

8.1 Deployed at Service Providers

The term "service provider" here refers to any organization that is making its digital resources, files, compute queues, instruments, etc., reachable, subject to access control, via XSEDE. This includes both the NSF service providers, as well as campuses, research groups, and individuals.

8.1.1 Genesis and UNICORE packages

Note that the following packages should only be deployed once the Genesis II XSEDE-specific configuration package has been deployed, as the latter defines both the trust store as well as the root of the namespace.

The UNICORE 6 Server package will run on Grid Interface Units at the Service Providers. The package implements the OGSA Basic Execution Services capability. It includes all JAR files to interoperate with Genesis II meta-scheduler and security environment.

The UNICORE 6 command-line client.

The Genesis II Server package includes the ability to share and instantiate all types of resources: compute, data, identity, and others.

The Genesis II Client package includes all client capabilities, the APIs, CLIs, and the GUI. It does not contain a server.

8.1.2 Grid Interface Unit

UNICORE 6 and Genesis II servers are deployed on what we call Grid Interface Units. A Grid Interface Unit (GIU) is a Windows, Linux, or MacOS host that can access resources the user wants to share. It is not necessary that the GIU be a dedicated host. However, external load may negatively impact the performance of the UNICORE 6 and Genesis II containers.

Recommend GIU Configuration

• Linux node - configuration
  ○ File system support for extended access control lists.
  ○ Local account that a grid administrator can ssh onto.
  ○ Ability to submit, stat, and delete jobs from the resource queues.
  ○ Require local queue configuration for the queues, e.g., queuing system used (PBS, SGE, etc), queue names, wallclock limits, properties of queues, special flags, etc.
  ○ Require local HPC node configuration information (number of cores, memory per node, number of nodes — basic stuff).

• 100 GB direct attached storage — speed matters, persistent state kept in on-disk database (more if to be used as a storage server).
  ○ Container uses an on-disk transactional database.
• Gigabit connection to sitewide file system and external network — prefer 10 gigabit.
• Prefer UPS power that outlasts transients and lasts as long as network still functioning.
• Public IP address
  o Use of GIUs behind a NAT is possible, but setup is more complex and will not be discussed here.
• If a user is going to share cluster or supercomputer resources, the GIU must share a file system with the compute nodes and it must have sufficient space for input and output files, as well as temporary files used by jobs.

Figure 85. Typical Grid Interface Unit configuration.
Some grid interface units can expect a heavy workload. For example, the frontends for the SP’s need to be both robust and fast to meet user expectations. If load becomes a problem, we propose that that SP grid interface units employ standard HA cluster approaches as shown below. Further, the incoming link should be full bandwidth to the external NIC, and ideally the nodes would be directly connected to the SP SAN.

### 8.1.3 Common components

A GridFTP server and MyProxy server. These two components can be deployed together via the Globus Connect Multi User package. See §6.3.2.
8.1.4 GRAM5 server
Details will be provided later.

8.2 Deployed as centralized services

(perhaps gffs-root.xsede.org)

- Genesis II XSEDE-specific configuration package — root of the namespace and Secure Token Services.
- XSEDE Training package on XSEDE portal. This includes training materials and documentation targeted at different user communities. The initial package consists of a tutorial targeted at end users and a tutorial targeted at systems administrators and others who need to install and manage Genesis II servers.
- XSEDE User Portal: see §3.1.1.
- Globus Online services: see §6.

8.3 Deployed on clients

The Genesis II client, UNICORE Command-Line Client (UCC), and UNICORE Rich Client (URC) may be deployed on client machines.

XUAS services (§6) require that client machines have a Web browser and/or secure shell client installed. In addition, the Globus Connect package (§6.3.1) is required if a user wants to transfer data to/from the client computer.
References

4. GT 5.0.0 GRAM5: User’s Guide. [Accessed June 1, 2012]; Available from: http://www.globus.org/toolkit/docs/5.0/5.0.0/execution/gram5/user/.
5. GT 5.0.0 GridFTP: User’s Guide. [Accessed June 1, 2012]; Available from: http://www.globus.org/toolkit/docs/5.0/5.0.0/data/gridftp/user/.
20. Box, D., Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., Kaler, C., Langworthy, D., Leymann, F., Lovering, B., Lucco, S., Millet, S., Mukhi, N., Nottingham, M., Orchard, D., Shewchuk,


25. Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S. Web Services Description Language (WSDL) 1.1. 2001; Available from: \url{http://www.w3.org/TR/wsdl}.


52. Li, V. Interoperability Experiences with the OGSA® WSRF Basic Profile 1.0 Open Grid Forum, 2009.
86. WS-I Basic Security Profile 1.0, 2007.