Enabling Scientific Workflow and Gateways using the standards-based XSEDE Architecture

Shahbaz Memon and Morris Riedel
Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH
Jülich, Germany
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, Iceland

Helmut Neukirchen and Matthias Book
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, Iceland

Andrew Grimshaw and Daniel Dougherty
Department of Computer Science
University of Virginia
Charlottesville, USA

Peter Kascuk, Márton István and Ákos Hajnal
Laboratory of Parallel and Distributed Systems
Hungarian Academy of Sciences, Budapest, Hungary

***Abstract*—The XSEDE project seeks to provide "a single virtual system that scientists can use to interactively share computing resources, data and experience." The potential compute resources in XSEDE are diverse in many dimensions, node architectures, interconnects, memory, local queue management systems, and authentication policies to name a few. The diversity is particularly rich when one considers the NSF funded service providers and the many campuses that wish to participate via campus bridging activities. Resource diversity presents challenges to both application developers and application platform developers (e.g., developers of gateways, portals, and workflow engines).**

The XSEDE *Execution Management Services* (*EMS*) architecture is an instance of the Open Grid Services Architecture EMS and is used by higher level services such as gateways and workflow engines to provide end users with execution services that meet their needs. The contribution of this paper is to provide a concise explanation and concrete examples of how the EMS works, how it can be used to support scientific gateways and workflow engines, and how the XSEDE EMS and other OGSA EMS architectures can be used by applications developers to securely access heterogeneous distributed computing and data resources.

***Index Terms*—Scientific computing, workflow, distributed computing, XSEDE, gateways, architecture.**

I. Introduction

The Extreme Science and Engineering Discovery Environment (XSEDE) project seeks to provide "a single virtual system that scientists can use to interactively share computing resources, data and experience."[1] The XSEDE system software architecture is use case driven. Each use case describes both the core functional requirements of the use case as well as the required quality attributes, e.g., that a particular service must respond to queries within one second. There are a set of use cases that appear again and again as components of more complex use cases, e.g. authenticate, run a remote job, transfer data. We call these use cases the canonical use cases. We believe these use cases are not unique to XSEDE, rather they are common across any wide area execution environment. The canonical use cases are designated as UCAN X, where UCAN stands for canonical use case, and X is the number. Currently, there are 12 canonical use cases. Unsurprisingly, many of the use cases involve computation in one form or another. The first "canonical" use case, UCAN 1 is "Run a Remote Job" [1]. Campus Bridging [2] has three computational use cases: CBUC 5 "Support for distributed workflows spanning XSEDE and campus-based data, computational, and/or visualization resources", CBUC 6 "Shared use of computational facilities mediated or facilitated by XSEDE", and CBUC 7 "Access to private cyberinfrastructure resources on a service-for-funds basis." Others include the High Throughput Computing use cases, the High Performance Computing use cases, and the Federation and Interoperation use cases. (These later three have not yet been formally published.) In order to understand how XSEDE implements the use cases one can take one of two approaches using documentation targeted at two different audiences. The first approach is to look at the use cases from an end-user perspective. The second approach is to look at the XSEDE architecture documents and what are known as the XSEDE architectural response documents. The XSEDE Architecture Level 3 Decomposition document [3] is the master architecture document. It describes all of the software protocols, interfaces, and interaction paradigms

This document was developed with support from National Science Foundation (NSF) grant OCI-1053575 and partly supported by NordForsk as part of the Nordic Center of Excellence (NCoE) eSTICC (eScience Tools for Investigating Climate Change at High Northern Latitudes). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF and NordForsk.

[1]www.xsede.org, accessed 31 July 2017

of the core architectural components. This document, often referred to as the L3D, is regularly updated and has been thoroughly reviewed by XSEDE stakeholders including, but not limited to, operations, security, and Software Development and Integration (SD&I).

The L3D document contains a description of the core XSEDE Execution Management Services (EMS) architecture. It also includes an abstract description of how third party gateways and workflow engines can utilize the XSEDE EMS infrastructure. The contribution of this paper is to provide a concise explanation and concrete examples of how the EMS works, how it can be used to support scientific gateways and workflow engines, and how the XSEDE EMS and other OGSA EMS architectures can be used by applications developers to securely access heterogeneous distributed computing and data resources.

The remainder of the paper is organized as follows. In Section II we present the XSEDE architecture web services and authentication/delegation mechanisms that underlie the Execution Management Services architecture. We then proceed to give concrete examples of how the architecture is used to support a simple grid queue (Section III), a DAGMAN-like workflow engine (Section IV), the WS-PGRADE/gUSE portal and workflow engine [4] (Section V), and Apache Airavata [5] Section VI which is used in scientific gateways. In Section VII we discuss related work and conclude with a discussion of deployment status.

II. XSEDE Architecture Background

A. Three-Layered Architecture

The XSEDE EMS has a three-layer architecture with an access layer, a service layer, and a physical layer [6]. While the service layer is defined in terms of standard web service port-types (interfaces), the access layer, that part of the architecture that defines how clients interact with the system, is not.

The access layer mechanisms to interact with the XSEDE EMS and authentication services include, but are not limited to, graphical user interfaces (GUIs), command line interfaces (CLIs), application programming interfaces, and file systems interfaces. For most of our discussion here we will use the CLI to illustrate interactions because they are easier to read and come with less syntactic baggage. We refer to the XSEDE L3D architecture [3] and the Omnibus Manual [7] for more information about more technical APIs as well as direct interaction capabilities via a virtualized file system.

B. Web Service Basics

The XSEDE EMS is based on service oriented architecture using a set of standard XML-rendered data structures and interfaces. Access to the XSEDE EMS services is via web services using the OGSA WSRF Basic Profile 1.0 [8]. The OGSA-BP in turn uses the Web Services Interoperability profiles, including the WSI Basic Security Profile [9]. What this means is that interaction with XSEDE EMS services is done using an interaction pattern realized using SOAP over HTTPS that essentially represent XML-based Remote Procedure Calls (RPCs).

Embedded in the SOAP header is a *credential wallet*, i.e. a set of identity tokens. The identity tokens represent both individual user identities as well as group membership and role assertions. Several identity token types are currently supported: username/password and signed Security Assertion Markup Language (SAML) [10] assertion chain being the two most frequently used today. In the near future OAUTH2 tokens [11] will also be supported.

While web services are used for client-service and service-service interactions, it is important to note that end users and application developers are unlikely to ever see a web services interface, because this involves dealing with complex XML structure, or understand the intricacies of properly inserting identity tokens into SOAP headers. This is critical as XML/-SOAP is not for human consumption.

C. Authentication, Authorizaton and Delegation

As described above the XSEDE's main data and compute services such as the Global Federated File System (GFFS) and EMS use security tokens embedded in the SOAP headers for authentication purposes. These tokens are typically signed SAML assertion chains. Credential wallet items can be used by the service to make authorization decisions. For example, the service might look them up in an authorized user file (e.g., a *gridmap* file), or use an access control list to determine authorization.

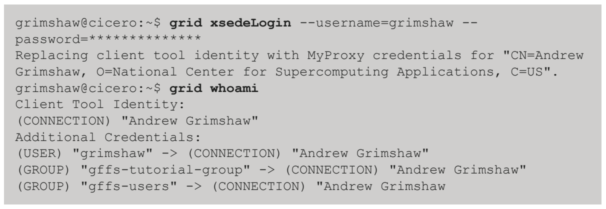
Providing secure authentication alone is not sufficient for many use cases. Suppose for example that the client requests that a broker perform some action, say scheduling a job on an execution service, on its behalf. Simply passing authentication tokens is insufficient unless they are bearer credentials. Similarly, if the execution service in turn wants to stage data in or out on behalf of the client that represents a key challenge.

To address these use cases, the EMS and GFFS security model support the notion of identity delegation. A user U1 may delegate to service S1 the right to perform actions on U1's behalf. Similarly, S1 might further delegate to S2 (as in our above example where the broker may further delegate to an execution service so that it can stage files.)

We accomplish this using a pre-delegation protocol in which clients pre-delegate to services the credentials in their credential wallet that they want the service to be able to use on their behalf.

From a programmer's perspective simply authenticate once to the XSEDE identity resource via the CLI or API and this will set the credential wallet. Figure 1 depicts an example of the CLI usage.

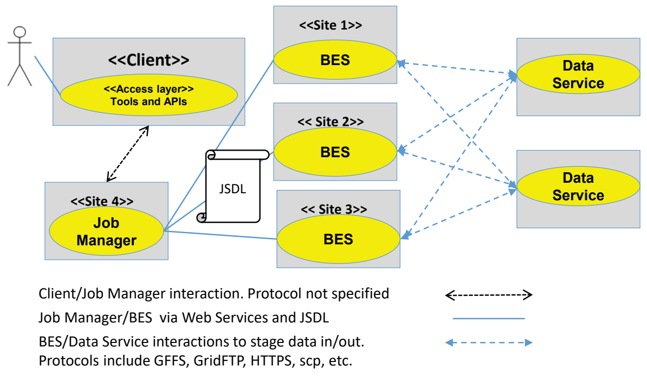
In the example, user *Andrew Grimshaw* has authenticated and received a MyProxy end-entity certificate to be used as a session certificate. The *grimshaw* identity as well as two group identities have been delegated to the session certificate. Additional credentials, e.g., group credentials, can be acquired at any time, and individual items of the credential wallet can be deleted.



```
grimshaw@cicero:~$ grid xsedeLogin --username=grimshaw --
password=**************
Replacing client tool identity with MyProxy credentials for "CN=Andrew
Grimshaw, O=National Center for Supercomputing Applications, C=US".
grimshaw@cicero:~$ grid whoami
Client Tool Identity:
(CONNECTION) "Andrew Grimshaw"
Additional Credentials:
(USER) "grimshaw" -> (CONNECTION) "Andrew Grimshaw"
(GROUP) "gffs-tutorial-group" -> (CONNECTION) "Andrew Grimshaw"
(GROUP) "gffs-users" -> (CONNECTION) "Andrew Grimshaw
```

Fig. 1. Demonstration of the CLI showing a user authenticating to the XSEDE identity resource.

The credential wallet (security context) is persisted to disk in the directory $GENII_USER_DIR. Thus, a program such as a gateway can keep multiple separate identities in different directories and simply change GENII_USER_DIR before each CLI call to select the appropriate security context. Similar tools are available in-memory in the *API.XSEDE Execution Management Services (EMS)*.



<<Client>>
<<Access layer>> Tools and APIs
<<Site 1>>
BES
<< Site 2>>
BES
<< Site 3>>
BES
<<Site 4>>
Job Manager
JSDL
Data Service
Data Service
Client/Job Manager interaction. Protocol not specified
Job Manager/BES via Web Services and JSDL
BES/Data Service interactions to stage data in/out. Protocols include GFFS, GridFTP, HTTPS, scp, etc.

Fig. 2. Clients interact with the job manager via some un-specified protocol. JMs interact with BES using standard protocols. BESes stage data in/out using a variety of protocols.

To provide a layer of virtual homogeneity for execution management at the access layer, XSEDE uses Open Grid Forum (OGF) job management specifications and profiles. The OGSA 1.5 Architecture Description [12], [13] and OGSA ISV Primer [14] provide good descriptions of Execution Management Services (EMS). Parts of the EMS architecture section come directly from the OGSA 1.5 Architecture Description and the XSEDE Architecture Level 3 description.

The EMS architecture is illustrated in Figure II. The user interacts with an access layer *(Client)* to run jobs, plan and execute workflows, or specify the computation they need. The client in turn interacts with a *job manager* (JM). The JM in turn interacts with *information services* (optional, not shown) and standard OGSA *Basic Execution Services* (BES) [15]. JMs specify the activities (single jobs or parameter sweep jobs) they want the BESes to execute using standard *Job Submission Description Language* documents (JSDL) [16]. JMs interact with the BESes via standard interfaces using web services.

D. Execution Management Services (EMS) Components

Execution Management Services are concerned with the problems of instantiating, and managing to completion, units of work, so called jobs to be executed on a cluster, 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:

1) 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, available libraries, and available licenses.
2) Selecting execution location. Once it is known where a unit of work *can* execute, the service must determine where it *should* execute to optimize some objective function.
3) Preparing for execution. Preparation could include deployment and configuration of binaries and libraries, staging data, or other operations to prepare the local execution environment.
4) Initiating the execution. Once everything is ready, the execution must be started.
5) 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.

The solution to these five problems consists of a standard job description mechanism and the use of set of services that decompose the EMS problem into multiple, replaceable components that all enable specific architecture functions. Job Submission Description Language [19] documents describe jobs, OGSA Basic Execution Services (BES) [18] to discover resource properties and execute jobs, GFFS directory paths [5] and resource registries (L3D section 5.2.3) to discover resources, and job managers to implement application-specific functionality. Below we briefly expand on each.

JSDL [16], [17]: JSDL is a standard XML-based language used to describe jobs. A JSDL 1.0 document has three main components: a resource requirements section, an application information section, and a data staging section.

The JSDL *resources section* contains information on application requirements such as operating system version, minimum amount of memory, number of processors and nodes, wall clock time, file systems to mount, and so on. It consists both of a standardized set of descriptions, as well as an open-ended set of matching requirements that are arbitrary strings.

The JSDL *application information* section includes items such as the command line to execute, the parameters, the job name, account to use, and so on.

The JSDL *staging section* consists of a set of items to stage-in before the job is scheduled in the local environment, and a list of items to stage-out post-execution. Each staging defines the protocol to use, the local file(s) to use as the source or target, and URIs for the corresponding source or target. Supported protocols include *http(s)*, *ftp*, *scp*, *sftp*, *GridFTP*, *mailto*, and the XSEDE *GFFS*.

A new version of JSDL is under development in the Open Grid Forum to address issues uncovered over the last several years. These include the ability to specify client-directed staging (as opposed to only server-based staging in 1.0), pre-and-post processing tasks to be executed in addition to the specified application, and additional resource descriptions to capture modern architectural features such as co-processors, e.g. GPGPUs, and detailed interconnection network requirements for large scale parallel jobs (e.g. use of torus topologies)!

A non-standards track extension, JSDL++ has also been developed to address the short-coming that each JSDL document describes exactly one set of possible resource matches with exactly one corresponding application execution description. For example, "the job requires 8 nodes, each with 8 cores, 64 GB memory, and MPICH 1.4: in that environment stage-in executable Y and execute 'Y 1024 -opt1' ". But what if an equally suitable option is " the job requires 1 nodes, each with 64 cores, 256 GB memory, and *pthreads*: in that environment stage-in executable Z and execute 'Z -opt2' "? JSDL++ allows the specification of an arbitrary list of options and the JSDL processing agent is free to use any one of the options for which it can find the resources.

OGSA Basic Execution Services (BESs) [15]: OGSA BES service endpoints represent the ability to execute jobs, specifically, to execute JSDL documents. The BES interfaces combined with JSDL create a virtual execution environment (EE) for XSEDE in which all execution resources, desktops, department servers, campus clusters, clouds, and supercomputers provide the same standard interface. It enables core functions of the XSEDE architecture.

The BES port types define both *Factory Attribute* and *Activity Management* interfaces. The Factory Attributes interface, *getFactoryAttributes()*, is used to discover the properties of the resource that the BES provides access to such as operating system, number of nodes, memory per node, and so on.

The Activity Management interfaces include *createActivity*, *getActivityStatus*, and *terminateActivity* porttypes.

CreateActivity takes as a parameter a JSDL document and returns (on success) a Web Services Addressing EndPoint Reference (EPR) [18]. The EPR is used as a handle to interact with the job. The BES is responsible for staging in data specified in the JSDL, starting and monitoring the job, keeping track of the exit code, and staging data out post execution.

getActivityStatus takes as a parameter the EPR of an activity created on the BES and returns the activity state (*Pending*, *Running*, *Running:Stage-In, Running:Stage-Out, Running:Queued, Running:Executing*, *Canceled*, *Failed*, and *Finished).*

terminateActivity takes as a parameter the EPR of an activity created on the BES and moves the activity to the *Canceled* state and cleans up any temporary files that may have been created.

Each of the above can operate on a single item, i.e., a JSDL document or an EPR, or on a vector of items. Thus, the execution environment consists of a set of BESs **EE** = $\{BES_0, BES_1, BES_2, ... BES_{N-1}\}$, each of which virtualizes a resource and implements the BES interface. Note that not all jobs can execute on all BESs, nor may all jobs have permission or allocation to execute on all BESs. Any given job being executed by a user may be executed on a subset of EE.

Access to the BESs is via the appropriate Web Services calls with authentication tokens carried in the SOAP header as described earlier, via the API, the GUI, the file system, or via the CLI.

```
grid run -jsdl=/path/to/jsdl/ls.jsdl /path-to-BES/besName
```

The first parameter is the path to the input JSDL file, either in the GFFS or in the local file system. The second parameter is the GFFS path to the BES on which to execute the job. The command is synchronous and will block till completion.

The asynchronous variant allows job status notifications to be stored on the GFFS space. The user can check on the status of the job by examining the status file. For the asynchronous execution user has to specify the -async-name attribute (*-async-name=/path/to/jobName*), an additional entity to the **grid run** statement mentioned above.

In the above, the command returns immediately after submission. The job's status is stored in the file specified by the grid path */path/to/jobName*. Eventually this file should list the job as FINISHED, FAILED or CANCELLED.

Performance: In a local environment the time for a client to call *grid run* is approximately *70mS*, in the wide area between 100-800mS. The vast majority of the time for local calls is in serializing/deserializing and validating the security context. Both BES implementations used in XSEDE can handle many concurrent calls. On an single core machine the Genesis II implementation can handle two or three concurrent callers without a performance degradation. On an eight core machine with sufficient memory (more than 4GB), the Genesis II implementation can handle approximately 20 concurrent calls.

Job Manager

The Job Manager (JM) sits directly above the BESs and information services and often sits between and mediate interactions between clients (end users or end user applications) and EMS services as shown in Figure II.

The JM is a higher-level service that encapsulates all 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 such as a science gateway or portal. The JM is the only intentionally unspecified, non-standard component of EMS, a condition that encourages the development of different styles and capabilities.

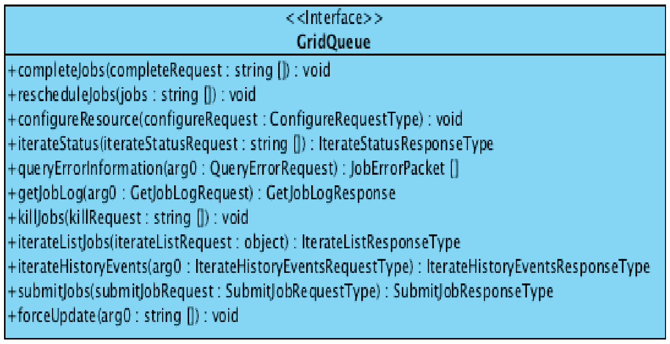
The JM is responsible for orchestrating the services used to start a job or set of jobs, by negotiating agreements, interacting with containers, and configuring monitoring and logging services. It may also aggregate job resource properties from the set of jobs it manages.

In XSEDE, two Job Managers are deployed: the *grid client GUI Create Job tool* and the *grid queue* service. There are also other JMs that are not following the XSEDE architecture, but still deployed in the XSEDE infrastructure, and are in use by different communities. They all follow the same pattern which we stated, but the details vary. It is our belief that many communities or tool developers may want to develop their own job management tools that are very customized to their environment and application requirements, but they can interact with the XSEDE EMS or XSEDE EMS compliant systems through standards-based interfaces.

To facilitate understanding in the sections below we briefly describe how Job Managers interact with BESes to perform their function: the Genesis II grid queue, the Genesis II DAGMAN workflow engine, the SCIBUS G-USE gateway and workflow engine, and the Airavata Gateway.

III. Simple Grid Queue

The grid queue interface is part of the GenesisII middleware, L3D §5.2.1.3 gives more comprehensive of view of its interfaces and capabilities. Each grid queue is configured to use a set of resources. Users submit jobs to the queue. The queue matches job resource requirements with BES factory attributes. Because sometimes jobs fail for no fault of their own, they are retried several times in order to provide an improved quality of service for users.



```
<<Interface>>
GridQueue
+completeJobs(completeRequest : string []) : void
+rescheduleJobs(jobs : string []) : void
+configureResource(configureRequest : ConfigureRequestType) : void
+iterateStatus(iterateStatusRequest : string []) : IterateStatusResponseType
+queryErrorInformation(arg0 : QueryErrorRequest) : JobErrorPacket []
+getJobLog(arg0 : GetJobLogRequest) : GetJobLogResponse
+killJobs(killRequest : string []) : void
+iterateListJobs(iterateListRequest : object) : IterateListResponseType
+iterateHistoryEvents(arg0 : IterateHistoryEventsRequestType) : IterateHistoryEventsResponseType
+submitJobs(submitJobRequest : SubmitJobRequestType) : SubmitJobResponseType
+forceUpdate(arg0 : string []) : void
```

Fig. 3. The GenesisII's Simple Grid Queue interface provides job-queue-like interfaces familiar to users of queuing systems, e.g., submit, kill, etc.

The queue is configured to use a subset of deployed BES resources using either the *qconfigure* command or the GFFS directory *ln* command. For each BES resource associated with a grid queue a maximum number of jobs that may be concurrently scheduled on the resource is set. This is called the number of *slots* for the BES. The grid queue keeps a list of available BESs and their associated *FactoryAttributes* and is used to match jobs to BESs.

Users submit jobs to the grid queue using either the BES *createActivity* interface, the queue *submitJobs* interface, or by copying a JSDL file into the *submission-point* pseudo-file. Whichever mechanism is used for submission, the result is the same. The job is added to the priority-ordered job queue. The job queue exists both on disk in a transactional relational database (for availability, reliability, etc.) and in an in-memory representation for performance.

When the JSDL specifies a parameter sweep job [19], e.g. used in bioinformatics [28], wherein a single JSDL file can generate tens to thousands of individual activities, the grid queue asynchronously expands the single JSDL into individual activities and places them in the RDBMS.

The information maintained in the database for each job includes the JSDL document, the serialized security context (i.e. the signed, delegated SAML chains), whether the job has been scheduled on a BES, the BES EPR and the activity EPR, and the number of times the job has been restarted.

The in-memory representation is much smaller. It includes the job name, the job owner certificates, the state, the name on the BES to which it is scheduled (if any), and the number of times it has been executed.

The grid queue scheduler is event-driven. There are three types of events: a job-arrival event, a BES status-change event, and a job-status change event.

A *job-arrival* event first stores the job in the database and then expands the job if it is a parameter sweep. Once safely stored, the grid queue scans the list of available BESs looking for matches between the jobs resource requirements and the BES factory attributes. “Available” here means that the queue has not submitted more than “slots” jobs to the BES.

A *BES-status-change* event occurs when either the number of slots for a BES is changed or periodic polling indicates that the BES is no longer accepting jobs. If the slots for a BES is increased the list of queued jobs is searched in order for a job that matches the BESs factory attributes. When a match is found the job is asynchronously started on the BES, and the scan continues until either all of the new slots are consumed or there are no more jobs to examine in the queue.

A *job-status* change event causes an update in the job status in the in-memory and on disk status. If the job has completed the in-use slot count for the BES is decremented, and if the in-use slot count is less than the slot count, a BES scheduling activity is started as described above. If the job has failed, the jobs' retries field is incremented. If it has reached a threshold it is marked as failed, otherwise it will be retried later. We also increment a failed job counter (that is aged) for the BES, and if it crosses the threshold we stop submitting jobs until it is back under threshold.

Note that job-status change events can happen one of two ways– either via periodic polling of job status using the BES's *getActivityStatus* method, or by asynchronous WS-Notification events sent by BESes that support notification subscriptions.

The key aspects of the grid queue example are the interaction patterns with both the client and XSEDE services. The client interacts with the grid queue using the Web Services interfaces, the GFFS, or the CLI. The grid queue interacts with XSEDE BESes via Web Services: BES factory attributes for resource discovery, BES Activity Management interfaces to start and manage jobs, and WS-Notification for asynchronous job state change notification.

Performance: Grid queue performance is very similar to BES performance, approximately 70ms per call in a local environment. *qstat* operations can take significantly longer depending on the number jobs in the queue. qsubmit commands where activity is a JSDL parameter sweep also take about 70ms. However that only counts the time to submit the sweep. Expanding the sweep progresses at a rate that varies from 10 jobs/second on slow machines with very slow disks, to 50 jobs a second with fast machines and SSDs.

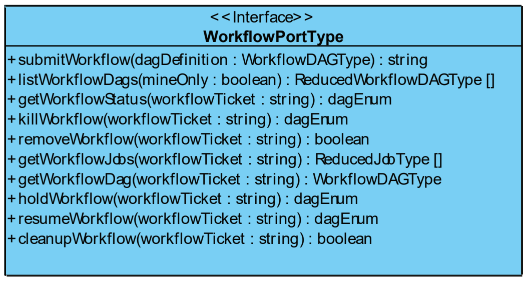
IV. DAGMAN-LIKE WORKFLOW ENGINE

Condor's DAGMan [20] uses a simple textual syntax to define workflows as Directed Acyclic Graphs (DAGs), where each node in the DAG represents a data transfer or computational task, or references another DAG to embed within the parent. DAGMAN is a well-known workflow graph representation mechanism and is used by scientists either directly or via other tools such as Pegasus [21] that are layered on top of it.

In DAGMan, jobs are defined using the Condor job description format [22]. XSEDE, on the other hand, uses the JSDL standard job description format, and uses the BES interface for job management on computing elements.

Our goal in the DAGMAN emulator for XSEDE was to execute DAGMAN-style workflows in XSEDE where job descriptions are in JSDL.

We defined a workflow non-standard execution engine interface known as a *WorkFlowPortType* which manages workflows defined using the same syntax as DAGMan. However, rather than using Condor job files as the vertices, WorkFLowPortType expects JSDL files[2]. Hereafter we will call instances of *WorkFlowPortType* work flow managers (*WFMs).*



<<Interface>>
WorkflowPortType
+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

Fig. 4. WorkFlowPortType interface. WorkFlowPortType instances are bound to a grid queue when they are created. In other words, they will submit all of their jobs to an associated grid queue which will do the actual job placement.

WFMs accept and execute workflow DAGs from the client via *submitWorkflow(WorkflowDAGType)* and return a ticket string (essentially a short GUID) that can be used to refer to the workflow. Workflows can be paused, removed, persisted, and resumed (see Figure 4 above).

WFMs can be instantiated on any Genesis II container (server), including containers on the clients machines in their lab, department, or university.

WFMs offer typical functions of workflow managers, they keep track of which jobs are running, which are ready to run, which have finished, and further basic functions.

Each WFM is associated when initialized with a grid queue using the pathname of the grid queue, e.g., */resources/xsede.org/queues/mainQ*. When a job becomes ready to run, the WFM submits the job to its associated grid queue and subscribes to job status notifications. To guard against notification failure the WFM periodically polls the grid queue for status. The WFM essentially delegates all aspects of job management to the grid queue, awaits notifications, and focuses on the work-flow-specific aspects of the problem.

[2]There is also a Condor-to-JSDL translator.

V. GATEWAY/WORKFLOW– SCIBUS/WS-PGRADE/GUSE

WS-PGRADE/gUSE is a science gateway framework that can be easily adapted by scientific user communities in order to create their own domain-specific gateway for XSEDE and other DCIs including major Grid types (Globus [23], UNICORE [24], gLite, ARC, BOINC), major cloud types (Amazon, OpenStack, OpenNebula) and major cluster types (Torque, SLURM, MOAB). In a similar way data access is extended to interact with major storage interfaces (HTTP, HTTPS, SFTP, GSIFTP, SRM, iRODS and S3) where large scientific data can be stored and processed by WS-PGRADE workflows. In WS-PGRADE workflows, nodes can be executed in any type of DCIs mentioned above and the workflow nodes at run time can access any types of storages mentioned above no matter of which type of DCI the workflow node is allocated and executed.

WS-PGRADE/gUSE [4] combines gateway and workflow technologies in a flexible way. WS-PGRADE presents a graphical user interface layer that supports the graphical creation of DAG-like workflows and provides visualization for workflow and job execution monitoring. gUSE is a high-level middleware layer that hides the low-level details of the underlying Distributed Computing Infrastructures (DCIs) and storage by using two major services: DCI Bridge for job submission [25] and Data Avenue for file transfers [26].

In terms of the EMS model of Figure 2 WS-PGRADE is acting as an access layer client tool and gUSE is playing the role of a job manager. WS-PGRADE and gUSE communicate via a proprietary non standard mechanism.

WS-PGRADE/gUSE was developed a European research project. More than 30 application-specific science gateways have been developed using SCI-BUS. The five layers of the SCI-BUS architecture are shown in Figure 5.

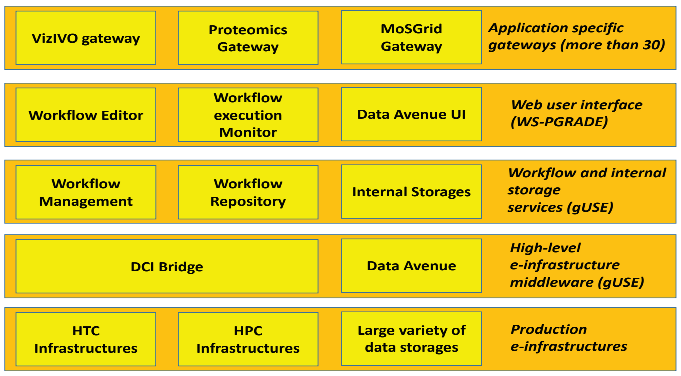

VizIVO gateway
Proteomics Gateway
MoSGrid Gateway
Application specific gateways (more than 30)
Workflow Editor
Workflow execution Monitor
Data Avenue UI
Web user interface (WS-PGRADE)
Workflow Management
Workflow Repository
Internal Storages
Workflow and internal storage services (gUSE)
DCI Bridge
Data Avenue
High-level e-infrastructure middleware (gUSE)
HTC Infrastructures
HPC Infrastructures
Large variety of data storages
Production e-infrastructures

Fig. 5. The SCI-BUS Architecture: Application specific gateways, WS-PGRADE UI, two gUSE layers and production e-infrastructures.

DCI Bridge is a generic job submission service that implements the OGF BES interface on top of various DCIs (mentioned above). Therefore any workflow system or gateway that uses the standard BES interface can submit jobs to all these DCIs via the DCI Bridge. Since the XSEDE EMS also uses the BES interface it was straightforward to integrate WS-PGRADE/gUSE and XSEDE via the DCI Bridge service.

In order to support the integration with XSEDE architecture, the DCI bridge's XSEDE extension plugin has been developed to facilitate the job submission and status monitoring scenarios. Below we describe more the integration approach employed and the functions supporting XSEDE requirements.

Integration Approach: The integration of a new type of DCI to WS-PGRADE/gUSE (such as one needed in XSEDE) requires the creation of a new plugin for the new DCI [26]. Henceforth we will refer to the new plugin as *XSEDE-plugin*. For *XSEDE-plugin* we extend the **Middleware** class via the **Plugin** class and override the necessary (abstract) methods. These methods are call-backs, called by the DCI Bridge on Web service calls (job submit, abort job), or periodically (query job status).

Job submission: Since the most typical job execution type in a scientific workflow is a parameter sweep execution, where the same job should be executed with many different inputs, we apply an optimization for calling the "grid" command to submit the jobs. This is particularly important since starting the XSEDE client by invoking the "grid" command is slow[3]. Therefore we execute the commands in "batch" mode. We collect the submit commands of the same user in a command file. Each user has a separate job submission file. Before submitting the user's job submission file through the "grid" command, the DCI Bridge XSEDE-plugin checks if a new job for the same user arrived at the DCI Bridge input queue. If it is arrived, the new job is added to the user's job submission file. After a short waiting time (100 ms) this check is repeated. If there are no more jobs of the same user in the DCI Bridge input queue the user's job submission file is submitted to XSEDE through the "grid" command that will execute the job submissions listed in the job submission file one after the other. If the number of job submissions reached a certain threshold (100 in the current implementation), the XSEDE-plugin immediately submits the user's job submission file to XSEDE.

Status request: The XSEDE-plugin requests the user job's status with the command "*./grid qstat queuename*". It returns the status of every job of the user stored in the resource queue so we do not need to request every job's status with separate command invocations. After this step, the XSEDE-plugin selects those jobs that have status "Finished" and with another single CLI call gets the results of these jobs.

Data staging: The data movement process is extremely easy thanks to the XSEDE architecture. The JSDL generated by gUSE contains the location of the input and output files needed for the job execution. The DCI Bridge XSEDE-plugin passes this information to standard-based middleware in XSEDE which takes care of data staging based on this information. This solution works only if the data storage has the protocol that is known by the XSEDE server. If this is not the case, the other service of gUSE called as Data Avenue can be used to realize data staging.

One of the key features that Data Avenue provides is its HTTP tunneling capability: For any remote file residing on a storage resource that Data Avenue supports an HTTP URL can be requested that can be read or written via simple HTTP GET or PUT operations by the clients. The created HTTP URL, called an *HTTP alias* points to the Data Avenue server, which will actually connect to the related storage resource at the time when the client initiates an HTTP GET/PUT operation. Bytes sent over HTTP PUT to the Data Avenue server are forwarded over the storage-related protocol by Data Avenue and written into the remote file on the remote storage. Similarly, bytes read from the remote file are forwarded as an HTTP stream in response to the GET operation of the client.

If user would like to use XSEDE as computation infrastructure, but with a data storage not supported by XSEDE, she should specify at workflow configuration time that a certain input or output file should be handled by the Data Avenue service, and also should provide the location of the file in the remote storage. Based on this information, the DCI Bridge can request an HTTP alias from the Data Avenue service for the given file. The DCI Bridge replaces the file staging information in the JSDL with the alias received from the Data Avenue service. The XSEDE server now can and will use this alias to access the file from the given remote storage using the HTTP tunneling function of Data Avenue service.

Note that PUT and GET operations are actually executed by the usual XSEDE data staging operations without modifying the XSEDE server middleware code after such an HTTP alias has been created. Notice that remote file contents can be read or written without additional authentication except the basic data exchange that is performed through a secure HTTP (SSL/TLS) connection. This kind of mediation service however requires allowing Data Avenue to connect to the storage resource on behalf of the user, thus the necessary credentials must be delegated while creating aliases. Depending on the authentication mechanisms, the credentials (such as passwords, X.509 proxies, secret keys) required by the storage has to to passed during the workflow submission stage. Thus, it is in most cases required to configure them before the submission. Furthermore, the Data Avenue deployments should strictly ensure that the credentials are not exposed by HTTP aliases in any way.

VI. Apache Airavata

Science gateways provide seamless and community driven interfaces for their often non technically-savvy users. As application-oriented interfaces provide domain specific interfaces with rich set of widgets and controls, science gateway developers prefer to choose a generic gateway framework for

[3]When this work was done the "grid" command took 4-10 seconds on every call, depending on platform. This was due to the load time of an extensive Java library stack. A new implementation, *fastgrid* starts a persistent process that takes 4-10 seconds only for the first time. Subsequent invocations take ˜70ms.

their users. Integration of these gateway frameworks use a set of client APIs to access backend computing and data services such as those provided in the above described XSEDE architecture.

Apache Airavata framework [5] is such a kind of framework that supports different scientific communities by providing connectors to various resources through which they can access data and run compute jobs. Since running a job is considered to be a vital element of the XSEDE architecture's L3D, we developed a dedicated connector for Airavata that enables the framework to access the resources with open standards which are the building blocks of the XSEDE architecture. Figure 5 illustrates the integration architecture of our implementation. The standards used are OGSA-BES and JSDL.

The specific point of our API integration is the Airavata GFAC component, which lies at the Airavata server side. GFAC is a meta-library that combines different kinds of computing and data services clients, such as Hadoop and Amazon S3 etc. The application runs are carried out as follows.

Initially, a user fetches her short lived X.509 credential from XSEDE's MyProxy service via her community user name and password token. The gateway client then uploads any data to a shared storage. In this case, the data transfer is achieved through UNICORE's storage management service, which is proprietary, but is based on the ByteIO and HTTP(s) standard data transfer protocols. Once the data is uploaded, a JSDL-compliant job request is constructed. The request contains the amount of resources required, application details, arguments, environment variables, and data staging points from where the data will be staged before and after execution. At this stage a job request has been prepared and the input data should be available at the execution site. The client then invokes the operation that tells UNICORE's standard OGSA-BES-compliant execution management service endpoint to run the job. The gateway client pulls the status until the job has completed. It then retrieves the application-generated output files and downloads them to the Airavata deployment. This output is stored for a gateway portal so that the gateway users can access it later.

Unlike the gUSE, which uses the Genesis II CLI to interact with the backend XSEDE BES services, the Airavata integration was carried out using the UNICORE 7 Java libraries to interact with the XSEDE backend BES implementation. The benefits of the tighter integration are faster service invocation, i.e., lower overhead. The disadvantage is the library requires the API users to integrate using Java.

Our implementation is deployed and currently used in production by the Ultrascan science gateway [27] community that spans the US and Europe. As one of the compute sites is the , it demonstrates the enormous capabilities of the XSEDE architecture to enable interoperability. In [27] we described more details on our implementation and demonstrated it using a production JURECA cluster located at Juelich Supercomputing Centre with real usage numbers.

The integration with the Airavata API will not only help the gateway instances of Ultrascan, but can serve a wide range of scientific disciplines intending to access resources through a uniform layer of abstractions. Hence, instead of using a CLI as described above, the access to XSEDE resources is given by an API that can be re-used in various other scientific gateways, workflow engines, or domain-specific GUI clients.

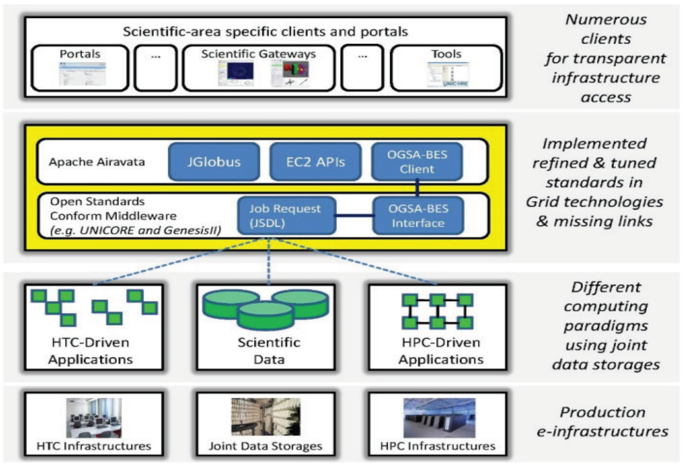

Scientific-area specific clients and portals
Portals
...
Scientific Gateways
...
Tools
Numerous clients for transparent infrastructure access
Apache Airavata
JGlobus
EC2 APIs
OGSA-BES Client
Open Standards Conform Middleware (e.g. UNICORE and GenesisII)
Job Request (JSDL)
OGSA-BES Interface
Implemented refined & tuned standards in Grid technologies & missing links
HTC-Driven Applications
Scientific Data
HPC-Driven Applications
Different computing paradigms using joint data storages
HTC Infrastructures
Joint Data Storages
HPC Infrastructures
Production e-infrastructures

Fig. 6. Integrated Architecture showing Scientific-area specific clients like the Ultrascan scientific gateway, the Apache Airavata API, and standards based middleware services being part of the XSEDE architecture.

VII. Related Work

There is a rich literature in both computational grids and workflows. Computational grid technology (formerly Metasystems [28], [29] flourished in the 1990's and early 2000's. Systems such as Legion [30], Globus [23], UNICORE [24], gLite [31], NetSolve [32], and many others were developed to support remote execution across heterogeneous platforms and administrative domains. The basic capabilities in all of these systems were similar: execute some job with a particular set of parameters, perhaps with pre-and-post staging, using a particular set of credentials, on a remote host.

By the early 2000's, the concepts were well understood and standardization efforts for remote job execution began in the Open Grid Forum (then the Global Grid Forum) Open Grid Services Architecture (OGSA) working group and in the Job Submission Description Language (JSDL) working group. These and other working groups in the OGF brought together stakeholders from industry, academia, and government in to agree on the fundamental features of grids. The OGSA-BES standards are themselves layered upon a whole set of industry standards in Web Services, security, and metadata management.

The XSEDE Execution Management Services architecture described here is different from previous remote execution systems insofar that it is built on the OGSA-BES, JSDL, and other open standards as opposed to a proprietary architectures. The advantages of a standards-based architecture are stability in the interfaces and protocols (a benefit for developers), avoidance of vendor lock-in (a benefit for organizations), and ease of integration of standards-compliant components. Ease-of-integration due to standards was particularly the case when integrating gUSE with XSEDE. gUSE/DCI were using JSDL/BES internally for their communication. Extending them to integrate with XSEDE BES was trivial. With respect to avoiding vendor lock-in, within XSEDE we regularly use two different BES implementations within XSEDE.

Workflow tools have been around for decades. Simple tools such as scripting languages, e.g., *bash*, and build tools, e.g., *make*, are still extensively used. More sophisticated tools such as DAGMAN, Makeflow, Pegasus, Kepler, Taverna, Swift, and BPL are in widespread use as well. Further, just about every gateway or science package, e.g., Galaxy, has an embedded workflow engine.

Despite these similarities, standardization in the scientific computing community has been difficult. Several unsuccessful attempts were made in the Open Grid Forum. All failed due a lack of consensus on even how to scope the problem.

The XSEDE architecture explicitly avoids workflow standardization because there seemed to be no consensus. Instead, the architecture is designed to support a variety of workflow tools. One of the goals of this paper has been to test the hypothesis that the XSEDE EMS can support many different workflow tools. One of the lessons learned is that some workflow tools prefer to manage their own file transfers.

VIII. Conclusion and Future Work

The contribution of this paper is to provide a concise explanation and concrete examples of how the job management and submission in the form of XSEDE EMS works, how it can be used to support scientific gateways and workflow engines, and how the XSEDE EMS and other OGSA EMS architectures can be used by applications developers to securely access heterogeneous distributed computing and data resources. This was done by first laying out the core components of the XSEDE EMS architecture and then demonstrating how the components are used by higher level gateways and workflows.

The examples demonstrated the functionality via the use of the command line interface for clarity. While the command line interface is fully functional, using the CLI is not as fast as using libraries. For high-volume production use, the API available from UNICORE [24] is preferable.

The XSEDE EMS is based upon open standards developed and brought into production over the last ten years. Unsurprisingly, along the way the implementers and users of these standards have uncovered some minor gaps in the specifications. These additional JSDL/BES requirements include, client-centric staging, i.e., staging under control of the client rather than the BES support pre-and-post processing stages and states. Further, a consistent mechanism to interact with jobs/activities as first class endpoints and with the session directory of a running job/activity. These lessons have been taken back into the OGF standards process, where updated versions of JSDL, BES and other specifications are in their final stages of the process.

References

[1] I. Foster and et al., "XSEDE Canonical Use Case 1: Run a Remote Job," 2013.

[2] C. A. Stewart and et al., " XSEDE Campus Bridging Use Cases," 2012.

[3] F. Bachmann and et al., " XSEDE Architecture Level 3 Decomposition," 2012.

[4] P. Kacsuk and et al., "Ws-pgrade/guse generic dci gateway framework for a large variety of user communities," *Journal of Grid Computing*, vol. 10, no. 4, pp. 601–630, Dec 2012.

[5] S. Marru and et al., "Apache airavata: a framework for distributed applications and computational workflows," in *Proceedings of the 2011 ACM workshop on Gateway computing environments*, New York, NY, USA, 2011, GCE '11, pp. 21–28, ACM.

[6] F. Bachmann and et al., " XSEDE Architecture: Level 1 and 2 Decomposition," 2012.

[7] Chris Koeritz, "GenesisII Omnibus Reference Manual 10.9," 2016.

[8] I. Foster and et al., "OGSA WSRF Basic Profile 1.0," May 2006.

[9] K. Ballinger and et al., "WS-I, Basic Profile Version 1.0," http://www.ws-i.org/profiles/basicprofile-1.0-2004-04-16.html, [Online; accessed 31-July-2017].

[10] R. Monzillo and et al., "Web Services Security: SAML Token Profile 1.1," Feb 2006.

[11] D. Hardt and et al., "The OAuth 2.0 Authorization Framework," https://tools.ietf.org/html/rfc6749, [Online; accessed 31-July-2017].

[12] A. Savva and et al., "OGSA^{TM} EMS Architecture Scenarios, Version 1.0," April 2007.

[13] C. Jordan and H. Kishimoto, "Defining the Grid: A Roadmap for OGSA^{TM} Standards, Version 1.1," Feb 2008.

[14] S. Newhouse and A. Grimshaw, "Independent Software Vendors (ISV) Remote Computing Usage Primer," Oct 2008.

[15] I.Foster and et al., "OGSA Basic Execution Service (BES), Version 1.0," Nov 2008.

[16] A. Anjomshoaa and et al., "Job Submission Description Language (JSDL), Version 1.0," July 2008.

[17] A. Savva and et al., "JSDL SPMD Application Extension," 2007.

[18] D. Box and et al., "Web Services Addressing (WS-Addressing)," http://www.w3.org/Submission/ws-addressing/, [Online; accessed 31-July-2017].

[19] M. Drescher and et al., "JSDL Parameter Sweep Extension," May 2009.

[20] P. Couvares and et al., *Workflow Management in Condor*, pp. 357–375, Springer London, London, 2007.

[21] E. Deelman and et al., "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17 – 35, 2015.

[22] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," in *[1988] Proceedings. The 8th International Conference on Distributed*, Jun 1988, pp. 104–111.

[23] I. Foster and C. Kesselman, "Globus: a metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, 1997.

[24] A. Streit and et al., "Unicore 6 - recent and future advancements," *Annals of telecommunications*, vol. 65, pp. 757 – 762, 2010.

[25] M. Kozlovszky and et al., *DCI Bridge: Executing WS-PGRADE Workflows in Distributed Computing Infrastructures*, pp. 51–67, Springer International Publishing, Cham, 2014.

[26] Á. Hajnal, Z. Farkas, P. Kacsuk, and T. Pintér, *Remote Storage Resource Management in WS-PGRADE/gUSE*, pp. 69–81, Springer International Publishing, Cham, 2014.

[27] S.Memon and et al., "Advancements of the ultrascan scientific gateway for open standards-based cyberinfrastructures," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 13, pp. 2280–2291, 2014.

[28] A. Grimshaw, A. Ferrari, G. Lindahl, and K. Holcomb, "Metasystems," *Commun. ACM*, vol. 41, no. 11, pp. 46–55, Nov. 1998.

[29] L. Smarr and C.E. Catlett, "Metacomputing," *Commun. ACM*, vol. 35, no. 6, pp. 44–52, June 1992.

[30] A. S. Grimshaw and A. Natrajan, "Legion: Lessons learned building a grid operating system," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 589–603, March 2005.

[31] M. Cecchi and et al., "The glite workload management system," *Journal of Physics: Conference Series*, vol. 219, no. 6, pp. 062039, 2010.

[32] H. Casanova and J. Dongarra, "Netsolve: A network-enabled server for solving computational science problems," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 3, pp. 212–223, 1997.