SDIACT-107 GLUE2 Update Design Document

16 May 2014
Version 1.0

Table of Contents

A. Document History

Relevant Sections	Version	Date	Changes	Author
Entire Document	1.0	05/16/2014	Changes in response to design and security review.	Warren Smith
Entire Document	0.1	02/14/2014	Baseline	Warren Smith

B. Introduction

This document presents the design that will provide the functionality described in SD&I Activity 107 (SDIACT-107). In addition, the system described here will provide the functionality needed to satisfy Canonical Use Cases 7 and 11 from the XSEDE Architecture and Design group. These use cases describe the general publish/subscribe functionality needed by XSEDE.

B.1. Definitions, Abbreviations and Acronyms

AMQP: Advanced Message Queuing Protocol. A standard protocol for distributed messaging. [AMQP-0.9.1]

IPF: Information Publishing Framework. A software package used to gather and publish resource information. [IPF]

JSON: JavaScript Object Notation. A simple textual format for representing structured information. [JSON]

GLUE2: Version 2 of the GLUE specification of the Open Grid Forum. Consists of an information model and definitions of how that model is represented in different information frameworks (e.g. LDAP, XML, JSON). [GLUE2]

C. System Overview

The system consists of two main components: A publish/subscribe messaging system and a software package that gathers and publishes resource information. The messaging system is deployed centrally for XSEDE and will be operated by XSEDE. The information gathering/publishing software is developed and distributed by XSEDE for Service Providers to deploy on XSEDE resources.

Once the messaging system is in production and Service Providers deploy the information gathering/publishing software, Service Providers will be able to shut off the unsupported globus-wsrf service that is currently used to publish jobs and load information.

D. Behavioral Design

The system must perform the behaviors described below and which are illustrated in Figure 1.

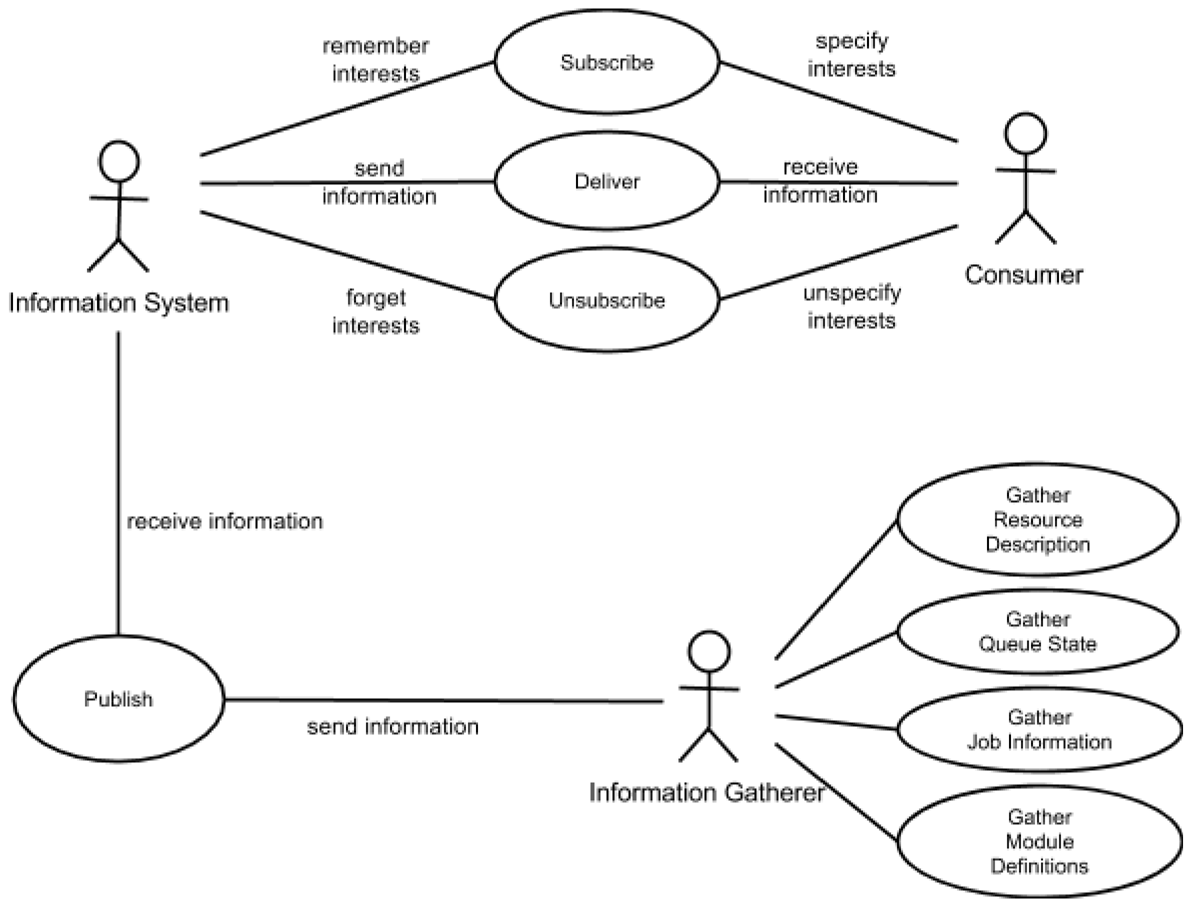

Figure 1. Use case describing the behavior of the system.

Gather Resource Description

The system will gather static and dynamic information about XSEDE resources. This static information includes node descriptions (e.g. number of cores, amount of memory) and node counts. The dynamic information includes node status (e.g. up or down) and node load (allocated, unallocated, or partially allocated).

Dynamic resource information should be gathered approximately every minute. Cluster scheduling systems often perform management activities every 60 seconds. Anecdotally, Interactive users check system status at most this frequently.

Gather Queue States

The system will gather descriptions of jobs being managed by the scheduling system of a resource, including the order in which the scheduler will consider pending jobs for execution. Job descriptions include enough information for users to track their jobs including submitting user, job name, job identifier, and job status (e.g. held, pending, running, complete).

Queue state information should be gathered approximately every minute. The queue state information is dynamic and is updated/accessed under the same reasoning as when gathering resource descriptions.

Gather Job Information

The system will gather information about individual jobs and their characteristics (e.g. submitting user, job name, job identifier, job status). This job information is gathered separately from overall queue state for users that only want information about specific jobs.

Job information should be continuously gathered. One important class of consumers of job information are tools that automate scientific workflows (such as science gateways). These consumers use job information as soon as it is known to trigger subsequent steps in workflows - the faster job information is known, the faster workflows (particularly workflows with many steps) will execute.

Gather Module Definitions

The system will gather information about the software available on XSEDE resources by examining the modules defined by either the ‘module’ package or the similar ‘lmod’ package.

Module information should be gathered approximately once an hour. Module information changes relatively infrequently (anecdotally, every few weeks) on stable resources. However, we do not want to provide incorrect module information for long periods of time.

Subscribe for Resource Information

A consumer of information describes the information that it wants to receive and the information system determines if it can deliver that type of information to this consumer. If it can be delivered (e.g. the user is authorized to receive it), the information system remembers this description and will use the description to determine which arriving messages should be delivered to the consumer.

Publish Resource Information

Once resource information is gathered on XSEDE resources it is published to the information system. This publication mechanism needs to be reliable and have sufficient performance to publish the volume of information gathered (as described above).

Deliver Resource Information
Resource information that is published to the messaging system must be reliably delivered to consumers. The consumers that receive any particular piece of information are determined using descriptions provided by consumers when subscribing.

E. Design Considerations

E.1. Assumptions and Dependencies

E.2. General Constraints

E.2.1. Hardware/Software Environment
The information gathering component must operate in the heterogeneous hardware/software environments of the various XSEDE resources. The information system operates in the virtual machines operated by SD&I to host information services.

E.2.2. End-user Environment
The environments used by consumers of this resource information are typically Linux systems, but the consumers can be written in a variety of programming environments such as Java (many science gateways), JavaScript (XSEDE portal services), and Python (Karnak queue predictor).

E.2.3. Availability or volatility of resources
XSEDE information services are supported 24x7 on a best effort basis by first-line XSEDE support and an on-call SD&I information services specialist. Individual server failures do occur and may not be corrected for several hours if they occur at an inopportune time. The design of this system should therefore automatically tolerate and recover from single server failures.

Individual XSEDE resources fail occasionally and the gathering component of this system must automatically recover and begin gathering and publishing information.

Network connectivity between an XSEDE resource and a subset or all of the XSEDE information system servers may fail. If connectivity to all XSEDE servers fails, the information gatherers must resume publishing when connectivity to any of the servers is restored. If connectivity to a subset of the XSEDE servers fails, the information gathers must continue to publish to the servers that are reachable. If network connectivity between the information system servers and the consumer is lost, the information system should either drop resource information or buffer resource information for later delivery, depending what the consumer requested during when subscribing.

E.2.4. Standards compliance
This design adopts appropriate standards including the Advanced Message Queuing Protocol (AMQP) and GLUE version 2.

E.2.5. Interoperability requirements
There are no specific requirements for interoperability at this time. However, one of the benefits of using the standards listed above is that it makes any interoperability we require in the future easier.

E.2.6. Interface/protocol requirements
None specified.

E.2.7. Data repository and distribution requirements
The information system must be able to buffer resource information, if requested by the consumer, to tolerate faults described in section E.2.3 and to allow consumers to more easily handle failures in their services..

E.2.8. Security requirements
Job information includes the local username of the user that submitted the job. This information can be useful to an attacker and therefore must be protected from anyone not participating in XSEDE.

It is important that consumers of resource information receive valid information. Therefore, only authorized publishers (e.g. service providers) are allowed to publish resource information.

E.2.9. Memory and other capacity limitations
The information gatherers should not use excessive memory (over 100MB) or processor cycles (more than an average of 0.2 cores) on XSEDE resources.

The amount of resource information that can be buffered by the information system is constrained by the amount of memory and disk available to the information system.

E.2.10. Performance requirements
Published information should be delivered to consumers that are ready to receive it within 1 second 99% of the time.

The information system must be able to handle the cumulative load of publishes of resource information and deliveries of resource information to consumers. Based on an analysis of XSEDE resources, resource use, and potential future expansion of XSEDE, the information system should be able to receive and deliver 150 pieces of resource information a second that require approximately 3 MB/sec of bandwidth.

E.2.11. Network communications
TCP/IP connectivity is needed between components of this system that supports a bandwidth of at least 3 MB/sec.

E.2.12. Verification and validation requirements (testing)

A number of different types of testing will be required including:

- Operation of information gatherers on the variety of XSEDE resource.
- Correctness of the resource information produced by the gatherers.
- That the overall system meets the specified performance/capacity goals.
- That the system responds to faults as required.

The test plan for this activity will provide further details.

E.2.13. Other means of addressing quality goals

An initial version of this system has been deployed across XSEDE as part of a pilot project. This pilot allowed long-term analysis and testing of the approach and has allowed potential users to use the pilot and provide feedback.

F. System Architecture and Detailed Design

The architecture of this system consists of three main components:

- A messaging service to deliver resource information. We have selected RabbitMQ as described in Section F.1.
- An information gathering framework to provide a software infrastructure to more easily gather information. We will use the Information Publishing Framework that has been previously developed and is described in Section F.2.
- Information gathering workflows (executed by IPF) to gather and publish (via the messaging service) the resource information that we wish to provide to consumers. We will use workflows that are part of IPF for generating resource information in the GLUE2 model as described in Section F.3..

The deployment of these components is shown in Figure 2 and will be further described below.

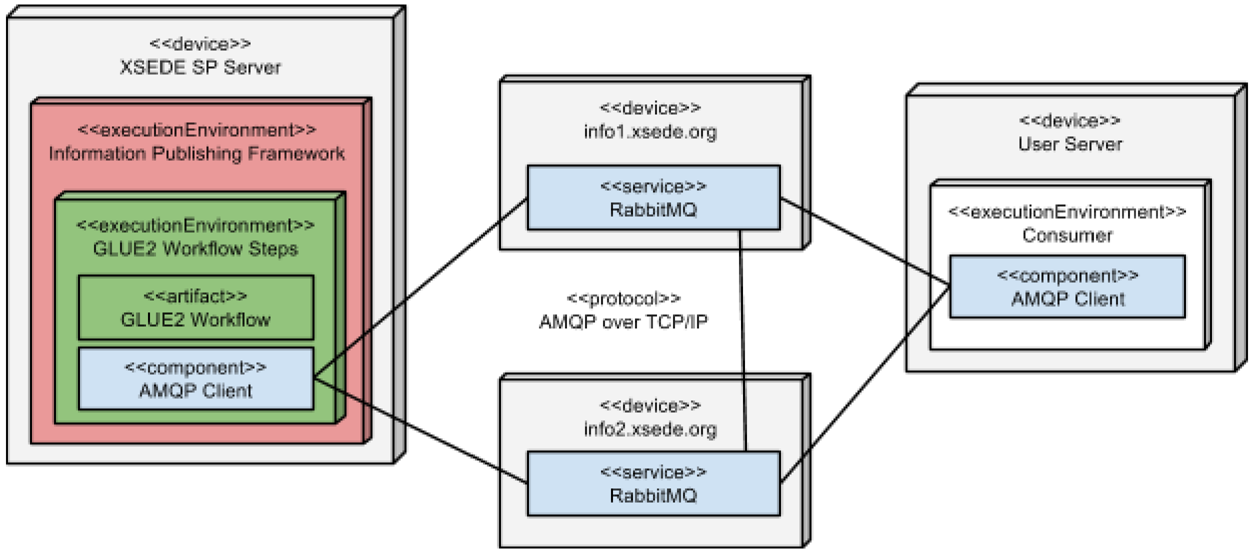

Figure 2. Deployment diagram of the system. Blue boxes are related to the messaging service, red boxes are IPF, and green boxes are related to GLUE2 information gathering workflows..

F.1. Messaging System

We have selected RabbitMQ [RabbitMQ] as the implementation of our messaging system for a number of reasons.

- It implements the AMQP 0.9.1 [AMQP-0.9.1] protocol - a standard protocol that is implemented by several different servers and a wide variety of clients. This allows us to switch to a different server implementation if needed and gives users many choices of how to interact with the service.
- It is high-quality software with a variety of features (such as a management web interface and SSL support) that provides high performance and high availability. The documentation is also high quality and the software is widely used.
- It is open source (Mozilla Public License).
- It is actively developed with available commercial support.
- It has a large user community which is actively supported by the developers (e.g. the developers have made several SSL-related improvements based on our suggestions).

RabbitMQ provides command line and web interfaces for administration. The functionalities are similar with the usual tradeoffs between such interfaces. The command line interface can be scripted for automated configuration (e.g. initial setup, managing users and permissions using information from XSEDE). The web interface presents information graphically (e.g. usage) and can be used without having to remember different commands and options. The ‘rabbitmq’ Linux user on the server hosting a RabbitMQ service can access all of the command line programs without additional authentication. The web interface requires that the administrator provide a username and password for a RabbitMQ account with administrative rights - multiple administrative users can be specified. The web interface is accessed securely over HTTPS.

There are a number of AMQP 0.9.1 and RabbitMQ concepts that are important to understand for this discussion. These are shown in Figure 3 and described below:

- A **virtual host** (vhost) is a logical container of other entities. Policies such as access control lists apply to a specific vhost.
- A **message** is an atomic piece of data that is received by RabbitMQ and ultimately delivered to the consumers that wish to receive it. A message contains content - the actual information that is being transmitted. A message also has a **routing key** that helps determine which consumers should receive a message. A routing key is a short string and typically contains important pieces of information about the content of the message (e.g. resource name, job identifier)
- An **exchange** is a location to which messages are published. A virtual host can have multiple exchanges and each exchanges can be used for messages with a different type of content, making it easier for consumers to only receive the messages they are interested in. There are different types of exchanges, depending on the type of functionality desired. The only type used in this work is a topic exchange where the

routing keys of messages act as publish/subscribe topics and are filtered against to determine where they should be delivered.

- A **queue** holds messages before a consumer is ready to receive them. A queue is bound to one or more exchanges and messages flow from these exchanges to the queue. When a queue is bound to a topic exchange, it specifies a **filter** that is used to determine which messages published to the exchange should be routed to the queue. The filter is a regular expression and is applied to the routing key of each message published to the exchange.

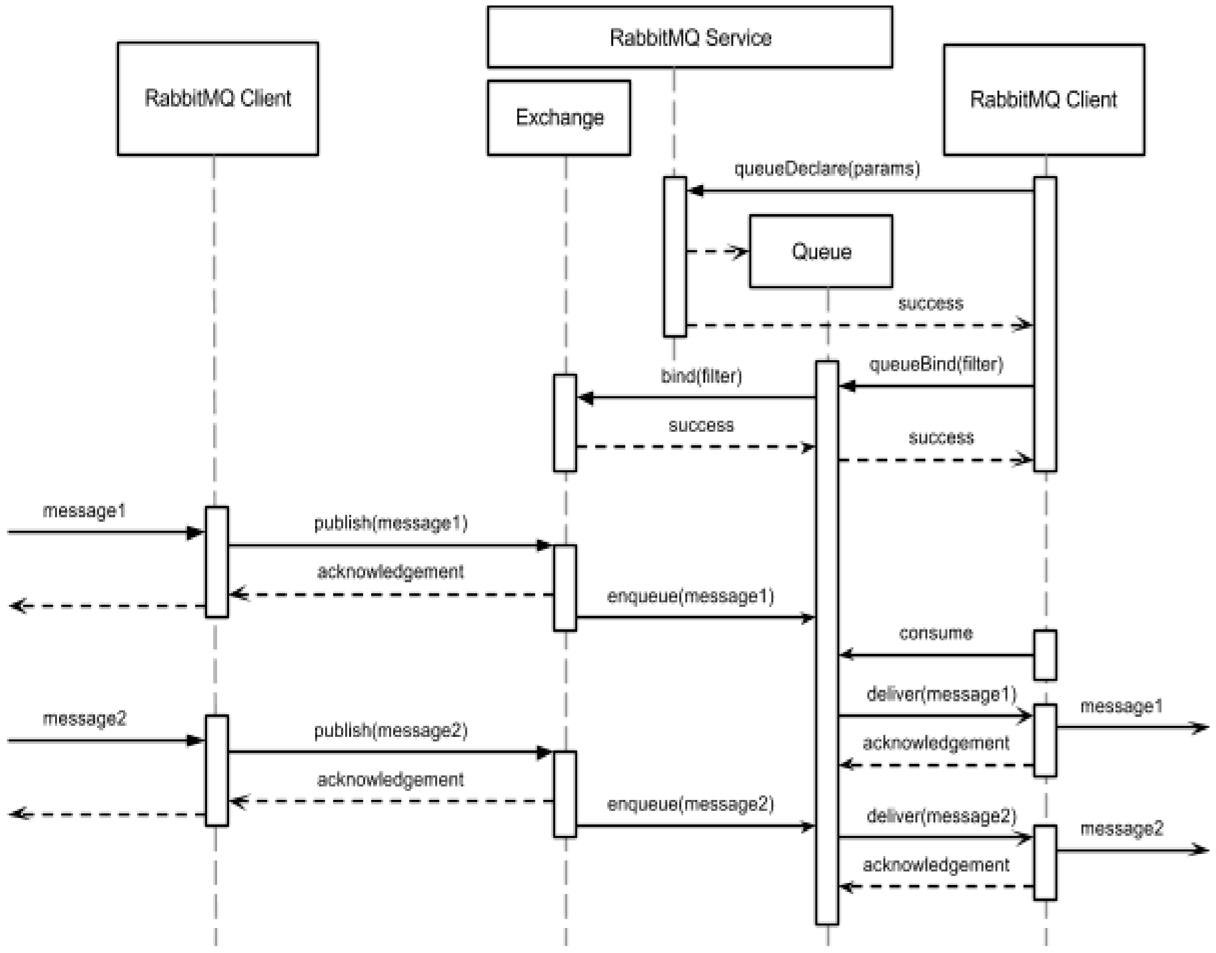

Figure 3. Sequence diagram showing a simplified version of how the messaging service transmits messages from producers to consumers.

There will be a single RabbitMQ virtual host ‘xsede’ used as a container for exchanges and queues. The following exchanges will be defined for GLUE2 messages:

- **glue2.compute** will be for messages that contain resource descriptions
    - The routing keys of messages will be <resource name>. For example: ‘stampede.xsede.org’

    - The content of each message will consist of the following GLUE2 entities (see [GLUE2] for the attributes of those entities and [GLUE2-JSON-examples] for example documents): ComputingManager, ComputingService, ComputingShare, ExecutionEnvironment, Location
- **glue2.computing_activities** will be for messages that contain queue states
    - The routing keys will be <resource name>. For example: 'stampede.xsede.org'
    - The content of each message will consist of a list of GLUE2 ComputingActivity entities. Activities that are waiting to execute should be listed in the same order as the local batch scheduler will consider them for execution.
- **glue2.computing_activity** will be for messages that contain information about individual jobs
    - The routing keys will be <job id>.<local user name>.<resource name>. For example: '12345678.johndoe.stampede.xsede.org'
    - The content of each message will consist of a single GLUE2 ComputingActivity entity.
- **glue2.applications** will be for messages that contain module descriptions
    - The routing keys will be <resource name>. For example: 'stampede.xsede.org'.
    - The content of each message will consist of a list of GLUE2 ApplicationEnvironment entities and a list of corresponding GLUE2 ApplicationHandle entities.

The names of these exchanges are based on the entity names in the GLUE2 specification that are transmitted through each exchange.

Authentication

RabbitMQ supports the SSL protocol and can be configured with a server X.509 certificate so that clients can authenticate the service and a secure connection can be established. RabbitMQ authenticates users via a username/password or via the Distinguished Name contained in a X.509 certificate presented by a client. There is a well known anonymous user with username 'guest' and password 'guest' that many client libraries will fill in by default.

The XSEDE RabbitMQ services will be configured with X.509 certificates so that they support secure connections and so that clients can authenticate them. The RabbitMQ services that are part of the pilot deployment use server certificates from InCommon.

XSEDE users will connect anonymously (as 'guest') or will provide an X.509 certificate to connect as that identity. Users will not be provided with passwords and will only be able to authenticate with certificates. The information that users will be able to receive in each of these situations is described below. Anonymous authentication will result in network communication being unencrypted over TCP. X.509 authentication will result in network communication being encrypted with SSL/TLS.

XSEDE Service Providers (SPs) will authenticate to the RabbitMQ services using an X.509 certificate. SPs will be publishing information and it is important to authenticate such publishers. This X.509 authentication will result in network communication being encrypted with SSL/TLS.

When XSEDE users and SPs authenticate with a X.509 certificate, that certificate must be issued by a Certificate Authority that is trusted by XSEDE. Note that at the time of writing, the InCommon Certificate Authority is not trusted by XSEDE. We assume that SPs will use host certificates for their resources when authenticating to publish information to RabbitMQ.

In the initial implementation, SPs and users will get access to the messaging system by submitting an XSEDE ticket that includes the DN from their certificate. These DNs will be manually added to RabbitMQ. We expect that the initial users will be science gateways and the user portal. The number of SPs and users will therefore initially be relatively small and this will be an acceptable approach. In the future, an option to consider will be automatically managing DNs in RabbitMQ using the DN information that is associated with each active XSEDE user account.

All authentication information will be stored in the Mnesia database used by RabbitMQ for this type information. A RabbitMQ administrator accesses this database via the rabbitmqctl command line program or the RabbitMQ web management interface.

Authorization

RabbitMQ authorization is based on configure, write, and read expressions associated with a user inside a virtual host. These expressions are regular expressions and specify the names of entities that a user can interact with. Configure permission lets a user create and delete exchanges and queues. Write permission lets a user write messages to an exchange or bind a queue. Read permissions lets a user retrieve messages from a queue or bind to an exchange.

Some of the messages through the system will contain information that must be protected. Table 1 describes the access controls that must be enforced for the types of information described in Section D and Figure 1:Two types of information can be public and two types must be restricted to XSEDE participants because they contain user names. If XSEDE user names are known to an attacker, an attacker can more easily perform password attacks or phishing attacks.

Table 1. Access control for different types of resource information.

Exchange	Information Type	Access	Notes
glue2.compute	Resource Description	Public	

glue2.computing_activities	Queue State	Private - XSEDE users and services only	Contains user names
glue2.computing_activity	Job Information	Private - XSEDE users and services only	Contains user names
glue2.applications	Module Definitions	Public	

The access controls described in Table 1 translate to the AMQP/RabbitMQ model as shown in Table 2. Consumers cannot write to GLUE2 exchanges and anonymous consumers cannot attach to GLUE2 exchanges that contain non-public information.

Table 2. Translation of access control to AMQP model.

Role	Publish to GLUE2 Exchanges	Subscribe to Public GLUE2 Exchanges	Subscribe to Private GLUE2 Exchanges
Information Gatherer	Yes	Yes, for testing of publishing	Yes, for testing of publishing
Authenticated Consumer	No	Yes	Yes
Anonymous Consumer	No	Yes	No

The exchange names selected for GLUE2 messages and the fact that RabbitMQ uses names ‘amqp.gen-XXXXX’ for queues that it generates for users leads to the access control lists in Table 3 below. The last row in the table shows the access control lists for a project that wants to create its own queues using their PROJECT name.

Table 3. RabbitMQ access control lists.

Role	Configure	Write	Read
Information Gatherer	^amq\.gen.*	^amq\.gen.* | ^glue2.*	^amq\.gen.* | ^glue2.*
Authenticated Consumer	^amq\.gen.*	^amq\.gen.*	^amq\.gen.* | ^glue2.*
Anonymous Consumer	^amq\.gen.*	^amq\.gen.*	^amq\.gen.* | glue2.compute | glue2.application

Authenticated PROJECT Consumer	^amq\.gen.* | PROJECT.*	^amq.gen.* | PROJECT.*	^amq\.gen.* | ^glue2.* | PROJECT.*

In the initial implementation, the access control for each user will be configured manually at the same time user authentication information (DNs) is set. This will be a manual process triggered by an XSEDE ticket and will be accomplished using a simple script that configures RabbitMQ correctly when given a (DN, role) pair. In the future, an option to consider will be automatically managing (DN, Authenticated Consumer) pairs in RabbitMQ using the information that is associated with each active XSEDE user account.

All authorization information will be stored in the Mnesia database used by RabbitMQ for this type information. A RabbitMQ administrator accesses this database via the rabbitmqctl command line program or the RabbitMQ web management interface.

Fault Tolerance

There are several different ways that RabbitMQ can be configured to support high availability. The approach we have selected is the simplest for administrators and users because it relies on RabbitMQ to automatically manage availability. RabbitMQ will be deployed on two XSEDE information servers in different locations. These servers historically fail several times a year due to issues with the servers (e.g. failures in the virtualization system), problems caused by services running on the servers (e.g. using all available disk space), or network failures. It is extremely rare for both servers to fail simultaneously.

The two RabbitMQ services will be **clustered** with each other [RabbitMQ-Clustering]. This is a RabbitMQ high-availability configuration where virtual hosts, users, access control lists, and exchanges are mirrored to all RabbitMQ instances. Essentially, the multiple instances act as a single virtual RabbitMQ instance. This approach is easy to administer, in comparison to federated or active/passive approaches that require administration of each instance. If an instance fails and then resumes execution, the RabbitMQ re-clustering will automatically synchronize the instances. If there is a network partition and the clustered instances operate separately, the RabbitMQ automatic re-clustering many not handle every case since there can be conflicts in the instances. It is therefore possible that an intervention by an administrator may be necessary to resolve those conflicts.

Clustering provides high availability of messages arriving at exchanges, but not necessary for delivery of messages to consumers. An example of this is if a queue is managed by a single RabbitMQ instance and that instance fails, the queue is not available until the node recovers. This situation may be fine for many consumers that don’t need to receive every message - they would just create a new queue on a different RabbitMQ instance and perhaps miss a few message. An example of such a consumer might be the user portal that is displaying current system state to users. However, other consumers may want to receive every message. An example of such a consumer is a science gateway that doesn't want to miss any state updates for jobs that it has submitted.

To ensure that consumers that want all messages do not miss any of them, we will allow the creation of highly-available queues [RabbitMQ-HA] that are mirrored across RabbitMQ instances. In this configuration, a message that is delivered to an exchange and is wanted by a highly-available queue will be stored on all of the mirrors of that queue on different RabbitMQ instances. If the RabbitMQ instance that a consumer is receiving messages from fails, the consumer can simply connect to a different RabbitMQ instance and continue receiving messages from their queue.

To take advantage of the clustered instances, clients described below in Section F.2 will be configured with the host names of each of the RabbitMQ instances. When a client starts up, it will randomly order the host names and then try to connect to each in turn. This approach is used instead of a DNS redirect, load balancer, or similar server-side approaches to reduce failures and complexity. A server-side approach that cannot detect when a RabbitMQ instance has failed would redirect clients to that failed instance, causing failures on the client side until the redirect is manually reconfigured. A server-side approach that can detect when a RabbitMQ instance has failed would be complex to implement.

F.1.1. Classification

Subsystem.

F.1.2. Definition

The messaging system transmits information from those producing it to those that want to receive it.
[The specific purpose and semantic meaning of the component.]

F.1.3. Responsibilities

The messaging system distributes messages containing resource information from where this information is gathered to the entities that wish to use this information.

F.1.4. Constraints

See Section E.2.

F.1.5. Composition

The major components of the Messaging System are shown in Figure 2 and consist of RabbitMQ services and clients. RabbitMQ services act as intermediaries: They receive messages published by clients, determine which consumers should receive each message, and then deliver those messages to consumer clients when those clients indicate they are ready to receive them.

F.1.6. Uses/Interactions
As shown in Figure 2, a RabbitMQ client is available as a step in the Information Publishing Framework. This step is used as part of GLUE2 workflows to publish the resource information gathered by those workflows. In addition, consumers of GLUE2 information (portals, science gateways, job scheduling/prediction services, etc.) use a RabbitMQ client to receive GLUE2 information.

F.1.7. Resources
The RabbitMQ services can run on servers with a range of resources. The amount of processing performed by a service is relatively low (one core is typically sufficient). The amount of network usage depends on the amount of message traffic. As indicated in Canonical Use Cases 7 and 11, 3 MB/sec of data through the service must be supported. The amount of memory and disk available to a RabbitMQ service constrains the amount of messages that can be buffered for later delivery. Approximately 1 GB of memory and 20 GB of disk has been sufficient during the pilot project, except for when consumers failed and stopped accepting messages for days at a time.

RabbitMQ clients generally use a minimum amount of resources, but their use of network bandwidth depends on the volume of messages they are sending and receiving. This volume will typically be under 3 MB/sec as mentioned above.

F.1.8. Processing
See Figure 3 and Section F.1 above.

F.1.9. Interface/Exports
Users interact with the RabbitMQ services using the AMQP protocol and the AMQP entities (virtual host, exchanges, routing keys) defined above. This is typically accomplished by using an AMQP client library. The programmatic interfaces of these libraries vary.

Administrators interact with the RabbitMQ services using a command line program (rabbitmqctl), the management web interface for the services, and the AMQP protocol and client libraries.

F.2. Information Publishing Framework
Gathering information about an XSEDE resource can be a complex task. There are typically multiple types of information that need to be gathered at different rates and to generate certain types of information, a number of gathering tasks need to be executed. In addition, there may be multiple ways that information needs to be published (e.g. AMQP, REST, write to a file) and consumers may want information formatted in different ways (e.g. JSON, XML).

IPF [IPF] is a system for gathering complex information that grew out of the glue2 software created in TeraGrid for gathering and publishing GLUE2 information. IPF generalizes the earlier glue2 software, can be used to gather additional types of information, and provides additional publishing mechanisms.

IPF executes information gathering/publishing workflows that are composed of simpler steps. The steps isolate the mechanisms of information gathering from the formatting and publishing of the information. The IPF framework is written and Python and IPF steps are Python classes. The following generic steps are currently available:

- Publishing: File, AMQP, HTTP PUT/POST
- Resource name: local hostname, 'tgwhereami'
- Site name: derived from local hostname, 'tgwhereami'
- Platform: from Python platform module, 'tgwhatami'

The process of preparing and executing a workflow is shown in Figure 4. The initial steps read the workflow and create a catalog of available steps. An IPF workflow is specified using a JSON document stored in a file. Available steps are stored in the lib/ subdirectory and are automatically discovered by IPF (classes that inherit from Step).

The next part of the process is to generate an instance of the workflow. This is done by creating an instance of all of the steps needed by the workflow and linking the inputs and outputs of these steps together. A helpful feature of IPF is that the workflow specification does not typically need to link steps together or even specify all steps. IPF will link steps based on the types of outputs generated and inputs needed, as long as there are no ambiguities (e.g. two steps producing the same type of output). Furthermore, if steps in the workflow require inputs that are not provided by any specified steps, IPF will examine the step catalog to locate steps that produce those inputs and add those steps to the workflow (again, as long as there are no ambiguities).

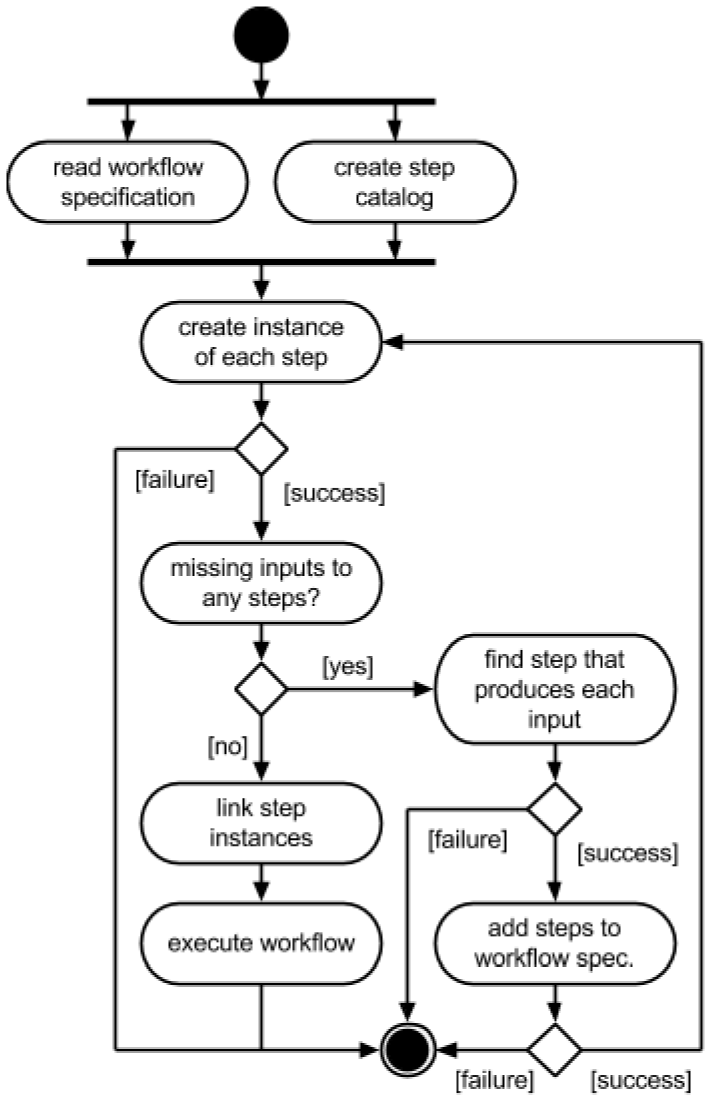
read workflow specification
create step catalog
create instance of each step
[failure]
[success]
missing inputs to any steps?
[yes]
find step that produces each input
[no]
link step instances
[failure]
[success]
execute workflow
add steps to workflow spec.
[failure]
[success]

Figure 4. Overview of the activities that construct and execute a workflow

Once instances of all specified and needed steps are created and linked together, the workflow is executed. Example GLUE 2 workflows are shown below in Figures 6-8 and the process to execute a workflow is shown in Figure 5. Each step is executed independently and when all of the inputs for a step are available, the step can begin to execute. If a step has no inputs, such as the leftmost steps in Figure 6, they can execute immediately. Once a step executes, its outputs are provided as inputs to other steps that require them. Once all steps either complete execution or fail, the workflow is complete.

One important feature to notice is that steps (and therefore workflows) can execute continuously. This is an important feature when a stream of information is being processed (e.g. job events in a scheduler log file such as in Figure 7). Some steps, such as log watchers, simply never complete execution and produce a number of outputs as they execute. Other steps, such as for publishing, execute to completion, but continue to wait for more inputs. A related feature is that IPF can determine when a step will not receive any more inputs and notify the step of this. The step can then decide if that is expected or if it should report an error. The error case occurs when a step fails and does not provide the inputs expected by other steps.

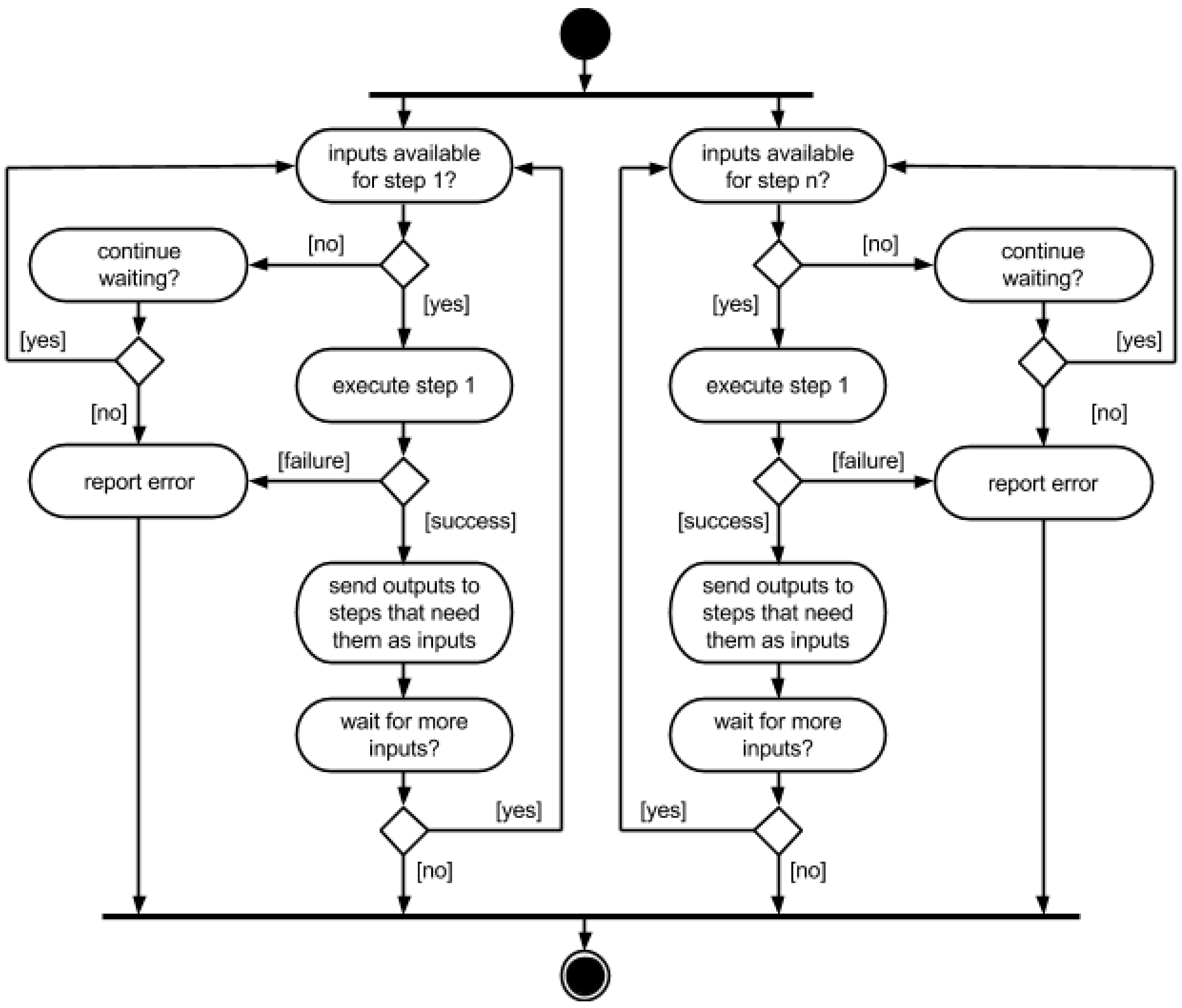

Figure 5. The activities that are performed when executing the steps of a workflow.

There are two ways to typical execute IPF workflows: periodically or continuously. Periodic workflows execute to completion (e.g. execute a qstat and publish the results such as in Figures 6 and 8) and can be run as cron jobs. Continuous workflows are daemons and execute for long periods of time (e.g. watch scheduler logs for job updates such as in Figure 7) and can either be started as part of operating system initialization or via cron jobs that periodically check that the continuous workflow is running and start it if it is not. IPF includes shell scripts for executing both of these types of workflows.

F.2.1. Classification
Subsystem

F.2.2. Definition
The Information Publishing Framework provides a software environment for executing complex information gathering and publishing workflows.

F.2.3. Responsibilities
IPF reads workflow specifications, prepares workflows for execution based on those specifications, and executes the workflows.

F.2.4. Constraints
None specified.

F.2.5. Composition
IPF can be considered a single component at this level of abstraction.

F.2.6. Uses/Interactions
IPF includes an AMQP publishing step which interacts with an AMQP client library. IPF includes steps and workflows for gathering GLUE2 information (described below).

F.2.7. Resources
IPF consumes processing cycles when it is executing workflows, but the amount of cycles it consumes should be low (perhaps 10% of a core) on top of the cycles consumed by the steps as they execute. IPF generates log files when it executes workflows and storage space is needed to store these logs. Additional storage space is needed the longer logs are stored and the higher the logging level.

F.2.8. Processing
See Section F.2.

F.2.9. Interface/Exports
There main interfaces that apply to XSEDE are:
- The shell scripts to execute workflows inside cron, as daemons, or from the command line. The interfaces for these are simple the script name and the name of the file containing the workflow specification.
- The JSON workflow definitions. Administrators of XSEDE may need to modify these definitions to be compatible with their local resource (e.g. setting resource names, paths).

F.3. GLUE2 IPF Workflows
A set of IPF steps and workflows are used to gather and publish resource information in the GLUE2 format. The following abstract GLUE2 steps are defined (the names correspond to entities defined in the GLUE2 specification):
- Activity, ComputingActivity
- Endpoint, ComputingEndpoint

- Service, ComputingService
- Manager, ComputingManager
- Resource, ExecutionEnvironment
- Application
- Compute (composite)

Concrete GLUE2 steps are needed for the schedulers and resource managers in use on XSEDE resources. At the current time, the following schedulers/resource managers are supported:

- Catalina
- Cobalt
- Condor
- LoadLeveler
- Moab
- Nimbus
- OpenStack
- PBS variants (including Torque)
- SGE
- SLURM

In addition, there are pre-defined workflows for the scheduler/resource manager combinations used by XSEDE resources. There are workflows to provide each type of information that needs to be gathered, as described in Section B.

Figure 6 shows the workflow for gathering a resource description and queue state for a system that is managed by Torque and Moab. We combine these into a single workflow because the dynamic part of the resource description (summary of jobs) is derived from the queue state. The top of the workflow diagram is for the queue state, the bottom is for the resource description, and at the very bottom is a step that discovers the name of the resource the workflow is running on. The resource name is used by many of the steps in the workflow so those links are not shown.

The steps related to queue state include a “Torque Jobs” step to execute the Torque qstat command and create a list of ComputingActivity (Job) objects, a “Moab Jobs” step to execute the Moab showq command to add priorities to the ComputingActivity objects and to sort the list of objects, a “Queue State” object to create the JSON document to publish and then two publishing steps to publish the queue state via AMQP and to a local file. Publishing to a local file is only for debugging purposes.

The steps for generating a resource description include “Torque Queues” that uses qstat to find information about queues, “Torque Nodes” that uses pbsnodes to find information about the nodes of the cluster, and steps that consolidate and summarize the information from these steps and from the queue state. The resource description document is published via AMQP for use by XSEDE and to a local file for debugging.

One thing to note is that the workflow shown in Figure 6 (and the workflows in Figures 7 and 8) have steps with ‘TeraGrid’ in their names. These steps use legacy TeraGrid tools (e.g. tgwhereami, tgwhatami) to get information about the resource. However, the information returned by these tools refers to XSEDE rather than TeraGrid. For example, the tgwhatami tool (used in the TeraGrid Resource Name and TeraGrid Site Name steps) on Stampede returns ‘stampede.tacc.xsede.org’ for the resource name and ‘tacc.xsede.org’ for the site name (during TeraGrid, these names would end with ‘teragrid.org’).

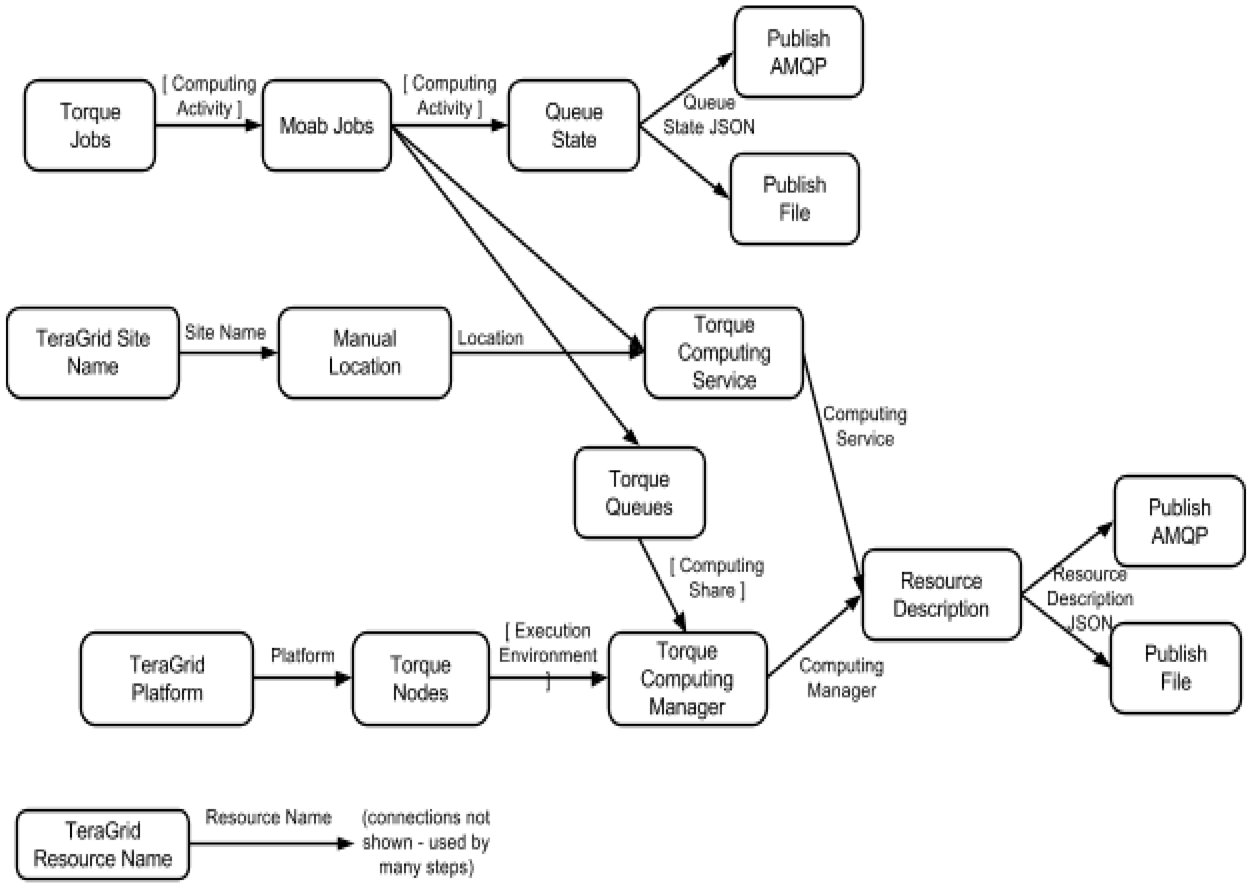

Figure 6. The workflow to gather and publish a resource description and queue state for a system managed by Torque/Moab.

Figure 7 shows the workflow that continuously gathers and publishes information about job state. This is primarily accomplished by a single step, “Torque Job Updates”, that continuously runs. This step watches Torque log files and when it sees jobs being submitted or changing state, it generates a ComputingActivity document for each individual job update. **Note that this workflow must run on a server that can access the scheduler log files and as a user that has permission to read them**. This step therefore sends a number of these documents to the “Publish AMQP” step, which will publish each document individually. In addition to getting job information from the Torque log files, the update step also runs qstat once for each job to get information about the job that isn’t included in the log file entries.

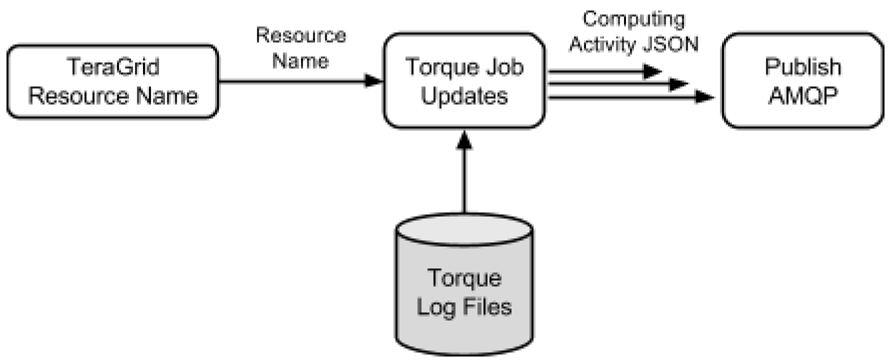

Figure 7. The workflow to continuously gather job information from Torque and publish it.

The final workflow is shown in Figure 8 and gathers and publishes information about the software installed on a resource by using module definitions. This information can be gathered from both modules and lmod and is accomplished by parsing the module files in the module path. The information to publish is an Applications document that contains a list of GLUE2 ApplicationEnvironment and a list of GLUE2 ApplicationHandle. An ApplicationEnvironment document describes a software package and an ApplicationHandle describes how to incorporate that application into a user environment. Each module results in one ApplicationEnvironment that has the name, version, description, and other information about the software in the module. Each module also results in one ApplicationHandle that has the string that a user uses to load that module into their environment.

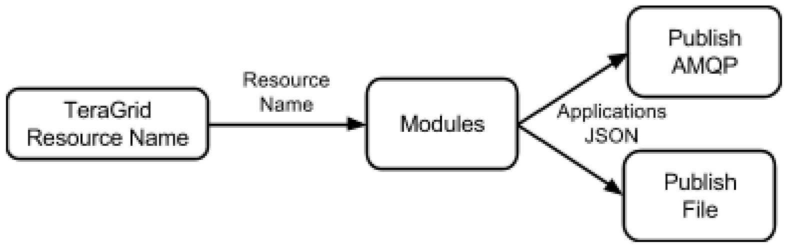

Figure 8. The workflow to gather and publish module information

F.3.1. Classification
Package

F.3.2. Definition
The IPF steps and workflows needed to gather XSEDE resource information in the GLUE2 format.

F.3.3. Responsibilities
This package must accurately gather resource information by interacting with XSEDE resources, particularly with the schedulers and resource managers of those resources.

F.3.4. Constraints
[Any relevant assumptions, limitations, or constraints for this component. This should include constraints on timing, storage, or component state, and might include rules for interacting with this component (encompassing preconditions, postconditions, invariants, other constraints on input or output values and local or global values, data formats and data access, synchronization, exceptions, etc.]

F.3.5. Composition
This package is composed of steps implemented in Python and workflow specification files as described in Section F.3.

F.3.6. Uses/Interactions
This package is part of the Information Publishing Framework and is used during execution of IPF workflows. The GLUE2 documents generated by the steps and workflows in this package are ultimately provided to XSEDE users and services.

F.3.7. Resources
Execution of the GLUE2 workflows requires processing to parse the information provided by schedulers and resource managers and to generate and publish the GLUE2 documents. Some types of workflows also publish to local files for debugging purposes, so a small (<10MB) amount of disk space is needed to store these

F.3.8. Processing
The processing is performed by IPF - see Section F.2.8.

F.3.9. Interface/Exports
See Section F.2.9 for the IPF-related interfaces. In addition, the GLUE2 workflows generate GLUE2 documents in JSON format that follow the JSON rendering [GLUE2-JSON] being finalized in the Open Grid Forum.

G. Interface Design

G.1. Interfaces With Other Systems
In addition to the user interface described below, the primary interfaces of this system are to the schedulers and resource managers that manage XSEDE resources. This system obtains the majority of the resource information that it publishes by querying the schedulers and resource managers on each XSEDE resource.

G.2. User interfaces
The users of this system are tools and services that interact directly or indirectly with XSEDE or XSEDE science gateway users. Examples include the XSEDE user portal, XSEDE science gateways, metaschedulers, and prediction services. The interface to these users consists of:

- The AMQP protocol and the AMQP client library selected by the user.
- The virtual hosts, exchanges, and queues configured in RabbitMQ.

- The format of the GLUE2 messages delivered via AMQP and the routing keys associated with those messages.

H. Packaging and Distribution

The software described in this document will be packaged as RPMs and distributed using the XSEDE yum/RPM repository.

There are a number of potential ways to divide this software into packages. The current approach is to divide it into the following RPMs for deployment on each XSEDE resource:

- A package ‘ipf’ which is the Information Publishing Framework. This package will include the IPF engine and basic, generic workflow steps.
- A package ‘mtk’ which is the Messaging ToolKit (MTK) [MTK]. This is an AMQP messaging library that is used by IPF to publish messages.
- A package ‘ipf-glue2’ which contains the GLUE2 publishing steps and workflows. This package will contain the steps and workflows for all of the different schedulers that are supported

The ipf-glue2 package will also include an interactive configuration script that will be executed as part of the RPM installation. This script will:

- Ask for the name of a user to execute the GLUE2 workflows. The workflows described here do not need to be run in a privileged account (e.g. root) and we recommend that they not be.
- Test if the tgwhereami and tgwhatami commands are available and ask if any environment configuration needs to be performed to access them (e.g. load a specific module). If they aren’t available, ask the installer to enter resource and site names.
- Ask which scheduler and resource manager are in use, if any environment configuration needs to be performed to access their commands, and where the log directory is.
- Generate an environment configuration script used by the workflow execution scripts.
- Generate workflow specifications appropriate for the information provided.
- Generate cron entries that will be printed to the screen and (if requested by the installer) added to the crontab of the user that will be running them.

I. References

[AMQP-0.9.1] Advanced Message Queuing Protocol 0.9.1.
http://www.amqp.org/specification/0-9-1/amqp-org-download

[UCCAN7] XSEDE Canonical Use Case 7: Subscribe for Resource Information.

[UCCAN11] XSEDE Canonical Use Case 11: Publish Resource Information.

[GLUE2] GLUE Specification v2.0. http://www.ogf.org/documents/GFD.147.pdf

[IPF] Information Publishing Framework. https://bitbucket.org/wwsmith/ipf

[JSON] Introducing JSON. http://www.json.org

[GLUE2-JSON] JSON Rendering of the GLUE 2.0 Specification.
https://github.com/OGF-GLUE/JSON

[GLUE2-JSON-Example] Example documents for the JSON Rendering of the GLUE 2.0 Specification. https://github.com/OGF-GLUE/JSON/tree/master/examples

[MTK] Messaging ToolKit. https://bitbucket.org/wwsmith/mtk

[RabbitMQ] RabbitMQ: Messaging that just works. http://www.rabbitmq.com

[RabbitMQ-Access] RabbitMQ Access Control. http://www.rabbitmq.com/access-control.html

[RabbitMQ-Clustering] RabbitMQ Clustering. http://www.rabbitmq.com/clustering.html

[RabbitMQ-HA] RabbitMQ High Availability. http://www.rabbitmq.com/ha.html