A RESTful Approach to the OGSA Basic Execution Service Specification

Sergio Andreozzi
INFN-CNAF
Viale Berti Pichat 6/2, I-40127 Bologna, Italy
Email: sergio.andreozzi@cnaf.infn.it

Moreno Marzolla
INFN Padova
Via Marzolo 8, I-35131 Padova, Italy
Email: moreno.marzolla@pd.infn.it

***Abstract*—The OGSA–Basic Execution Service (BES) specification has recently been proposed by the Open Grid Forum (OGF) as the standard job submission and management interface across different Grid middlewares. This specification defines a Web Service interface in terms of a Web Services Description Language (WSDL) document for creating, monitoring and managing computational jobs (called activities), and for querying the capabilities of the BES service itself. In this paper, we propose an alternate incarnation of the BES functionalities according to the Representational State Transfer (REST) architectural style. We describe the mapping of the BES operations in terms of HTTP actions on resources. We compare the REST formulation of BES with the standard WS-based one. We show that all BES operations can be expressed in a very natural way using the standard HTTP protocol and following the REST approach; moreover, we present useful extensions that are expected to appear in the near future.**

I. Introduction

The Grid paradigm emerged in the last decade for the integration, utilization and management of heterogeneous networked resources part of different administrative domains to be made available to virtual organizations [1]. Different middleware suites have been developed to support the Grid paradigm. They expose the various resources using a common abstraction layer offering a uniform access to them.

A job submission and monitoring service is one of the basic functionalities of most Grid systems available today. This service allows users to submit computational jobs to a Grid, manage them and monitor their progress. While the exact notion of "job" usually varies from Grid to Grid, there are many common features which can be isolated. For example, a "job" usually consists of running some executable program on a given processor; the program may operate on one or more input data files, and produce one or more output data files. Moreover, job requirements (minimum available memory, disk space, CPU speed) may be part of the job description.

Job management involves suspending, resuming, or removing a Grid job. Monitoring involves checking the current status of the job. Moreover, job submission services also provide operations to handle the service itself, e.g., disabling further job submissions or check the service capabilities. The different Grid middleware platforms offer different interfaces for job submission and monitoring services today. This makes interoperability between different Grids extremely difficult: jobs originating on a Grid system cannot directly be submitted to another Grid relying on a different middleware, both because the job description notation is different and because the interfaces to the job submission services are incompatible.

The OGF is the standard body which is defining specifications to enable interoperability both at the technological level and at the functional level. The overarching document defines the Open Grid Services Architecture (OGSA)[1] [1] in terms of a set of capabilities required to realize the Grid paradigm based on the principles of Service Oriented Architecture (SOA) and incarnated in the Web Services (WS) technologies.

Among the defined capabilities, the Execution Management Services are concerned with the problems of instantiating and managing to completion units of work. The related core specifications are the BES and the Job Submission Description Language (JSDL). JSDL is an XML-based notation for describing computational jobs [2], while BES is a WSDL-based interface for a Job Submission and Monitoring service [3]. In particular, the latter defines two WS port-types, which are shown in Table I with their corresponding operations. The *BES-Management* port-type is used to control the BES service itself, that is, starting and stopping the service. This port-type should normally be used by the system administrators. The *BES-Factory* port-type defines operations for creating and manipulating activities and set of activities. Moreover, it contains an operation (*GetFactoryAttributeDocument*) for retrieving attribute information about the BES service itself.

The BES *CreateActivity* operation returns an Endpoint Reference (EPR), which can be used by clients to refer to this activity. During the execution, activities traverse a number of states. The basic state model comprises the following states: (1) *pending*, the service has created the activity, but the execution is not yet started; (2) *running*, the activity is executing in some computational resource; (3) *finished*, the activity successfully completed the execution; (4) *terminated*, the activity has been terminated by calling the *TerminateActivity* BES operation; (5) *failed*, the activity has failed due to some error or failure. Finished, terminated and failed are terminal states. The state model can be extended to consider new states.

[1]OGSA, Open Grid Services Architecture, OGF and Open Grid Forum are trademarks of the OGF

TABLE I
BES PORT-TYPES AND OPERATIONS

BES-Management Port-type	
StartAcceptingNewActivities	Administrative operation: Request that the BES service starts accepting new activities
StopAcceptingNewActivities	Administrative operation: Request that the BES service stops accepting new activities
BES-Factory Port-type	
CreateActivity	Request the creation of a new activity; in general, this operation performs the submission of a new computational job, which is immediately started
GetActivityStatuses	Request the status of a set of activities
TerminateActivities	Request that a set of activities be terminated
GetActivityDocuments	Request the JSDL document for a set of activities
GetFactoryAttributeDocument	Request the XML document containing the properties of this BES service

The current state of art in Grid is characterized by middleware providers all adopting the SOA paradigm with particular incarnation in WS technologies. WS technologies are widely adopted, nevertheless they introduce a high level of complexity due to the richness of modeled functionalities and the availability of competing specifications. The interoperability is therefore still a challenging task because either different middleware suites rely on different set of WS specifications or because of incompatibilities about how the specifications are implemented. WS technologies are not the only possible incarnation of the SOA paradigm on top of the Web architecture. Another approach which is gaining popularity is the adoption of plain HTTP-based applications designed to comply with the REST architectural style. REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations [4]. Many distributed applications that successfully build on RESTful HTTP technology are today available, thus implying that WS technologies are not the only solution for Web-based distributed systems.

In this paper, we describe how the BES functionalities can be mapped into a RESTful HTTP-based approach. The goal is to show that this solution is viable and reduces the complexity of the considered service, while bringing in all the benefits of the REST-based approach.

II. RESTFUL BES

The REST architectural style was derived from the Web architecture and can be applied to different systems to obtain the following benefits: scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems [4]. Due to its origins, it is a natural application to distributed systems based on the HTTP protocol [5], nevertheless it can be applied also to WS-based distributed systems. The core architectural elements of REST are: (1) *Resource*, that is any entity which is needed to be identified; it is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time; (2) *Resource Identifier*, that is a Uniform Resource Identifier (URI) identifying a resource; (3) *Resource Representation*, that is data and/or metadata describing the current or intended state of a resource.

The main constraints posed by this architectural style are (see [4] for a complete list): *stateless*, each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server; *cache* the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable; if a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent request; *uniform interface* the operations identify only actions with a well-defined semantics and properties of safety and idempotency; no scoping information is provided in the operation name. In the remaining part of this section, we propose the mapping of the BES specification into the RESTful HTTP protocol, that is using the HTTP protocol with respect to the REST architectural style.

The methodology adopted to achieve this mapping consists of the following steps: (1) identify the interesting resources; (2) name the resources with URIs; (3) define the operations on the resources; (4) design the representations accepted from the clients; (5) design the representations served to the client; (6) define error conditions to be handled.

A. Modeling Resources and Resource Identifiers

We now present the definition of the resources that we consider useful in the RESTful BES; we also propose the URI structure for them (see Table II).

We let `/activities` represent the list of all activities present in the service; `/activities/`*`id`* is the current representation of a specific activity (`id` is the local identifier of the activity); `/activities/`*`id`*`/submitted` denotes the JSDL document which was used to instantiate the activity; `/activities/`*`id`*`/status` is the current status of the activity; `/` is the representation of the service capabilities (BES factory attributes document); `/status`] is the current status of the BES service; `/activities/`id_1`[;`id_j`]*` denotes the current representation of the activities identified by id_x; finally, `/activities/`id_1`/status[;`id_j`/status]*` is the current representation of the status of activities identified by id_n.

TABLE II
RESTFUL BES RESOURCES

`/`	representation of the service capabilities (BES factory attributes document)
`/status`	current status of the BES service
`/activities`	the list of all activities present in the service submitted to the given *share* (e.g., batch queue)
`/activities/`*id*	the current representation of activity *id* (*id* is the local identifier of the activity)
`/activities/`*id*`/submitted`	the JSDL document which has been used to instantiate activity *id*
`/activities/`*id*`/status`	the current status of activity *id*
`/activities/`id_1`[;`id_j`]*`	the current representation of the activities identified by id_n
`/activities/`id_1`/status[;`id_j`/status]*`	the current representation of the activity statuses identified by id_n

B. Modeling Operations

In this section, we describe how the BES operations can be mapped into standard HTTP/1.1 operations (GET, PUT, POST, DELETE) with respect to the REST constraints. The mapping of the WS-based BES operations onto the RESTful BES operations is summarized in Table III.

The BES specification defines operations that act not only on single activities, but also on set of activities (see Table I). In particular, *GetActivityDocuments*, *GetActivityStatuses* and *TerminateActivities* operate on a set of activities at the same time. This approach enables for instance to terminate multiple activities with a single BES operation invocation. This feature is particularly desirable as it reduces round-trip delays caused by multiple individual request/response interactions. It also allows the BES service to process operations more efficiently by batching them. When considering the mapping of the WSDL-based BES operations into HTTP operations, we need to take into account that the HTTP protocol operates on a single resource and does not support operations on a collection of resources. This issue is typically faced by defining a resource which maps to a set of entities (see Section II-A).

C. Modeling Representations and Status Codes

We now analyze each operation listed in Table III and describe the exchanged resource representations together with the involved status codes.

As regards the HTTP status codes, if not differently specified, we act as follows: for each operation, the server returns the `401 Unauthorized` HTTP response code if the user is unauthorized to access the *whole* BES service; this corresponds to the `NotAuthorizedFault` BES fault; for each operation involving the client sending an XML document in the request body (e.g., `PUT /activities/`), the server returns a `400 Bad Request` status code when the XML document in the request body is invalid.

Another general case to be considered is the one about operations working on multiple entities, for which individual status codes are needed (e.g., a terminate operation on an activity can be successful while on another can fail). The HTTP protocol does not provide native support for this case, in fact extensions were proposed to solve this issue (see WEBDAV specification and the multi-status code [6]). In this context, we prefer to act as follows: multiple response values are inserted into the HTTP response body. The HTTP `202 Accepted` status code will be issued by the BES service to denote that the request has been accepted and processed, and to signal the client that the results are contained in the response body.

- `GET /activities/`: this operation retrieves the list of all activities submitted by the caller which have not yet been removed. This operation has no equivalent in the BES specification. A `200 OK` HTTP response code denotes that the request completed without errors. In this case, the response body contains the list of URIs corresponding to the base path of each activity owned by the caller, rendered as a `text/xml` document with the following structure:

```
<activities>
  <activity>/activity/ID</activity>*
</activities>
```

- `PUT /activities/`: this operation requests the creation of a new activity. This is equivalent to the *CreateActivity* BES operation. The HTTP request body contains a BES `ActivityDocument` XML element as defined in [3]. This element basically contains a `jsdl:JobDefinition` sub-element which describes the structure and requirements of the activity being created [2]. The format of the request body is the following:

```
<bes:ActivityDocument>
  <jsdl:JobDefinition>
    ...
  </jsdl:JobDefinition>
  <xsd:any> *
</bes:ActivityDocument>
```

The HTTP response code can be one of the following:

- `201 Created` Upon successful creation of the activity, this status code is returned (the HTTP `Location` header will contain the URI for the newly created activity)
- `501 Not Implemented` This response code corresponds to the `UnsupportedFeatureFault` fault returned by the BES when it does not support some of the features requested by the JSDL; the HTTP response body should describe the features which are not supported
- `503 Service Unavailable` This response code corresponds to the `NotAcceptingNewActivities` fault returned by the BES if it is not accepting new activities.

Upon successful creation (HTTP return code `201 Created`), the `Location: URI` header is used to

TABLE III
RESTFUL BES ACTIONS

Resource	Operation	Description	BES counterpart
/activities/	GET	List all activities of the requester	none
	PUT	Create a new activity	*CreateActivity*
/activities/id_1[;id_j]*	GET	Get the current representation (JSDL document) of one or more activities	*GetActivityDocuments*
	DELETE	Remove (purges) one of more activities	none
/activities/id_1/status[;id_j/status]*	GET	Current status of a set of activities	*GetActivityStatuses*
	POST	Change the status of a set of activities (e.g., terminate the activities)	*TerminateActivities*
/	GET	Get the attributes of the BES service	*GetFactoryAttributesDocument*
/status	GET	Get the status of the BES service	*IsAcceptingNewActivities*
	POST	Change the status of the BES service (e.g., stop accepting new activities)	*SetAcceptingNewActivities*

return to the client the base URI of the newly created activity, as follows:

```
HTTP/1.1 201 Created
Location: /activities/ACT001
```

• DELETE /activities/id_1[;id_j]*: this operation is used to remove (purge) one or more activities from the BES service. The removal of the activities includes also the removal of all local files and directories that were generated by the activity itself. Note that the current BES specification does not provide any operation for purging a terminated activity. A 202 Accepted HTTP status code denotes that the request has been accepted. The response body will contain the detailed status information related to the removal of each individual activity. The response body is an XML document containing one <activity> element for each activity referenced in the request URI. If the <activity> element contains a <UnknownActivityIdentifierFault> element, then the activity was not found. Other kind of fault elements could be defined to notify the caller of other, implementation-related errors.

```
<deleteResponse>
  <activity id="id">
    <UnknownActivityIdentifierFault.../>?
  </activity>
</deleteResponse>
```

• GET /activities/id_1[;id_j]*: this operation gets the current representation of an activity, in the form of the JSDL document which describes the activity. This is equivalent to the *GetActivityDocuments* BES operation. Note that the current representation of an activity may be different from the original one. This is because the BES service might have processed and modified the original JSDL to reflect the current status of an activity. The original representation for activity *id* is thus accessible at the URI /activities/*id*/submitted. The 202 Accepted status code denotes that the request has been accepted. In this case, the response body contains an XML document with the following structure:

```
<ActivityDocumentResponses>
  <ActivityDocumentResponse>
    <ActivityIdentifier> uri </ActivityIdentifier>
    <ActivityDocument>
      {jsdl:JobDefinition}
    </ActivityDocuemnt> |
      <UnknownActivityIdentifierFault/>
  </ActivityDocumentResponse>*
</ActivityDocumentResponses>
```

The response document contains the URI and the JSDL document which was used to instantiate the activity or the current one (depending on the request). If the activity does not exist, the JSDL document is replaced by a <UnknownActivityIdentifierFault> element.

• GET /activities/id_1/status[;id_j/status]*: this operation retrieves the current status of a set of activities. This is equivalent to the *GetActivityStatuses* BES operation. With the Cache-Control: must-revalidate HTTP header, the user can request the BES server to ignore any cached status information, and explicitly check for the job status. The HTTP response code can be one of the following:

- 202 Accepted The operation has been accepted by the BES service; results are contained in the response body.
- 412 Precondition Failed The Cache-control: must-revalidate request header was supplied by the client, but the server does not support the possibility of explicitly polling the job status.

The BES server might use the Expires HTTP header to inform the client that the status information is valid until the next update. If the server is employing polling to query the status of the activities, than it might know the time of the next (possible) status update, and inform the client through the Expires header. If the response code is 202 Accepted, the HTTP response body contains an XML document with the following structure:

```
<ActivityStatusResponse>
  <ActivityStatus>
    <ActivityIdentifier> uri </ActivityIdentifier>
    <ActivityStatus>
      {bes:ActivityStateType}
    </ActivityStatus> |
```

```
      <UnknownActivityIdentifierFault.../>
  </ActivityStatus> *
</ActivityStatusRespose>
```

where <ActivityIdentifier> contains the URI of the activity (e.g., /activities/ACT001). If the operation was successful, the <ActivityStatus> element contains a child element of type ActivityStateType. In case of errors, the <ActivityStatus> element is replaced by <UnknownActivityIdentifierFault/>, which denotes that the activity ID does not exist. Both ActivityStatus and UnknownActivityIdentifierFault are defined as in the BES specification [3].

- POST /activities/id_1/status[;id_j/status]$*$: this operation changes the status of one or more activities. This is similar to the *TerminateActivities* BES operation, except that the REST counterpart would allow the user to request an arbitrary status change. This is useful in conjunction with specialized BES state models allowing for example an activity to be suspended/resumed at any time. Currently, the BES specification does not provide operations for changing an activity status, except for the *TerminateActivities*. The HTTP Request body is as follows:

```
<StatusChangeRequest>
  <ActivityStatus>
    <ActivityIdentifier> uri </ActivityIdentifier>
    <ActivityStatus>
      {bes:ActivityStateType}
    </ActivityStatus>
  </ActivityStatus> *
</StatusChangeRequest>
```

The 202 Accepted response code means that the operation has been accepted by the BES service; results are contained in the response body, according to the following structure:

```
<StatusChangeResponse>
  <ActivityStatus>
    <ActivityIdentifier> uri </ActivityIdentifier>
    <ActivityStatus>
       {bes:ActivityStateType}
    </ActivityStatus> |
      <UnknownActivityIdentifierFault.../>
  </ActivityStatus> *
</StatusChangeResponse>
```

For each successfully applied status change, the response document reports the URI of the activity with the new (updated) status; in case of failure, the URI is followed by an <UnknownActivityIdentifierFault> element.

- GET /status: this operation is used to check the status of the BES service, that is, whether the server is accepting new activities. This is equivalent to the *IsAcceptingNewActivities* BES operation. If the response code is 200 OK, the HTTP response body will contain the single XML element as follows:

```
<ServiceStatus status="open" | "closed"/>
```

where the status="open" attribute denotes that the service is accepting new activities, while status="closed" denotes that the service does not accept creation of new activities.

- PUT /status: this operation is used to change the status of the BES server. This operation is equivalent to the *StartAcceptingNewActivities* and *StopAcceptingNewActivities* BES operations.
The request body contains a single <ServiceStatus> XML element, with attribute status="open" to denote that the service should (re)start accepting new activities, and status="closed" if the service should refuse creation of new activities.

```
<ServiceStatus status="open" | "closed"/>
```

- GET /?schema=*S*: this operation is used to request the capabilities of the BES service. In its simplest form, (GET /) it returns the capabilities of the BES service as an XML document of type BESResourceAttributesDocumentType as described in [3]. This document contains a summary of the capabilities of the BES service (number of contained resources, operating system name, number of CPUs and so on). If the BES implementation supports additional resource models (allowed by the specification [3]), the client can access the alternate resource descriptions by using the ?schema=*S* query string, possibly combined with the Accept: HTTP header to specify the resource rendering format. For example, to get an XML rendering of a GLUE resource description [7] the client can issue this request:

```
GET /?schema=glue HTTP/1.1
Host: bes-service.example.org
Accept: text/xml
```

To get a text rendering of the same resource, the client would issue the following request:

```
GET /?schema=glue HTTP/1.1
Host: bes-service.example.org
Accept: text/plain
```

III. Extensions and Security Considerations

Some optional extensions are described in the BES specification. In this section, we describe how these extensions can be implemented in a RESTful way; we also make some considerations about the security infrastructure which can be used to authenticate interactions with the service.

A. Idempotent Execution

The BES specification allows an optional extension to support *idempotent execution semantics*. This extension can be used to ensure that issuing a *CreateActivity* request multiple times for the same activity results in the creation of at most one instance of the activity. It requires that a user-generated request ID should be associated to the *CreateActivity* request, so that the BES server can ignore duplicate requests.

Idempotent Execution can be implemented within the standard HTTP protocol in different ways. The Post Exactly Once (POE) protocol [8] works by having the server generate a unique URI for a POE resource, which is then used by the client to perform the actual POST operation. Should the server receive a duplicate POST for the same URI, it will return to the client a 405 Method Not Allowed. While this approach

has the advantage of being almost completely transparent to client applications (no need to send any special HTTP header), it requires an additional request-response iteration for the client to get the unique URI to user for the POST request.

Another solution would be that of inserting a client-generated unique ID in the HTTP request `Pragma` header, as follows:

```
POST /activities/ HTTP/1.1
Host: bes-service.example.org
Pragma: IdempotentActivityID=client_defined_id_01
...
```

As in the POE protocol, we let the service return a `405 Method Not Allowed` if it receives a duplicate request. The response header will also contain a `Location` field with the complete URI of the existing (already created) activity.

B. Lifetime Management

The *Lifetime Management* extension allows the client to request a specific maximum lifetime for an activity. After the lifetime expires, the server is allowed to remove the associated activity without further notice. Similarly to the Idempotent Execution extension, the maximum resource lifetime can be defined with an appropriate HTTP header in the request message, as follows:

```
POST /activities/ HTTP/1.1
Host: bes-service.example.org
Pragma: InitialTerminationTime=<datetime>
...
```

If the server, for any reason, is unable to comply with the requested activity lifetime, it will reply with a `400 Bad Request` error code. After the expiration of the activity lifetime, subsequent attempts to access the resource generate a `410 Gone` HTTP status code, denoting that the server permanently removed the requested activity.

C. Security Considerations

Although security considerations are outside the scope of the BES specification, they play a fundamental role in the actual deployment and usage of Web-based services. The use of HTTP over TLS/SSL [9] allows the clients and the RESTful BES service to mutually authenticate using digital certificates. In particular, this allows the service to ensure that only the owners can access the resources.

Another security issue may arise when the BES service needs to access remote data on behalf of the user, for example, when the submitted JSDL includes data staging elements. In this case, it is often not appropriate that the BES service authenticates as itself with respect to the remote storage service; for this reason, many Grid systems employ the so called *credential delegation* based on RFC3820 proxy certificates [10]. Credential delegation is a two-step process: first, the client (delegator) asks the service (delegate) to create a public-private key pair and uses it to generate a Certificate Signing Request. Then the delegator signs the certificate with his private key, and sends the signed certificate to the service. The delegation process can be implemented in a RESTful way very easily, as sketched here:

1) First, the user executes a `GET /delegation` operation to request the Certificate Signing Request.
2) The signed certificate is sent back to the service with a `PUT /delegation/`*id* request. Here, *id* denotes a user-defined delegation ID, which can be later used by the client to refer to this delegated credential (e.g., by inserting it into an appropriate HTTP header).

IV. Conclusions

In this paper we considered the BES specification, which is the standard interface to be adopted by the different Grid middlewares for interoperable job management. The specification, in its current form, relies on WS technologies, in particular it is described using a WSDL grammar. We presented a mapping of the BES functionalities into a REST-based approach using the HTTP protocol. We showed that both the core BES functionalities and the optional extensions can be implemented in a REST compliant way. The REST approach is generally considered simpler and easier to implement than the WS-based counterpart; moreover, REST services could in principle be tested using any HTTP client (for example an ordinary Web browser) without the need to develop specialized client applications. Future work will expand the security considerations (they were intentionally left out from the BES specification [3], but are nevertheless fundamental for any real-world implementation), and the development of an actual RESTful BES prototype.

References

[1] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich, "The Open Grid Services Architecture (OGSA), version 1.5," OGF GFD-R.80, Jul 2006.

[2] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva, *Job Submission Description Language (JSDL) Specification, Version 1.0*, Nov. 2005, OGF GFD-R.056.

[3] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer, *OGSA Basic Execution Service, Version 1.0*, Nov. 2008, OGF GFD-R.108.

[4] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, 2002.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol–HTTP/1.1*, Jun. 1999, RFC 2616, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[6] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen, *HTTP Extensions for Distributed Authoring–WEBDAV*, Feb. 1999, RFC 2518, `http://www.webdav.org/specs/rfc2518.html`.

[7] S. Andreozzi, S. Burke, F. Ehm, L. Field, G. Galang, B. Konya, M. Litmaath, P. Millar, and J. Navarro, "OGF GLUE 2.0 Specification," OGF Proposed Recommendation in Public Comment.

[8] "Post Once Exactly (POE)," Mar. 19 2005, `http://www.mnot.net/drafts/draft-nottingham-http-poe-00.txt`.

[9] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol, Version 1.2*, Aug. 2008, RFC 5246, `http://tools.ietf.org/html/rfc5246`.

[10] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson, *Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile*, Jun. 2004, RFC 3820, `http://www.ietf.org/rfc/rfc3820.txt`.