GWD-I (Proposed Informational)
OGSA HPC Profile WG

Marvin Theimer, Microsoft
Chris Smith, Platform Computing
Marty Humphrey, University of Virginia

July 1, 2006

HPC Job Scheduling: Base Case and Common Cases

Status of This Document

This document provides information to the Grid community on batch job scheduling of scientific/technical applications, also broadly referred to as the “core” high performance computing (HPC) use case. It does not define any standards or technical recommendations. Distribution is unlimited.

Copyright Notice

Abstract

This document describes the set of use cases for batch job scheduling of scientific/technical applications, also broadly referred to as the core high performance computing (HPC) use case. A simple base case is defined that we expect to have universally implemented by all batch job scheduling clients and schedulers. Additional "Common Cases" are enumerated, which are anticipated to be applicable to at least two but not all batch job scheduling clients and schedulers. This Base Case and Common Cases will be used as a set of requirements for the forthcoming OGF proposed recommendation entitled "OGSA™ HPC Profile". This document is a product of the OGSA HPC Profile WG of the Open grid forum.

Contents

1. Introduction

The objective of the OGSA HPC Profile Working Group (http://forge.gridforum.org/projects/ogsa-hpcp-wg) is to work on the profile and specifications needed to realize the vertical use case of batch job scheduling of scientific/technical applications. This use case is often referred to as the "core" high performance computing (HPC) use case. In recognition that batch job scheduling is only part of the larger design space of execution management services and that there are both simpler and more complex forms of batch job scheduling – some of which still have open research questions attached to them – the approach that will be taken by the OGSA HPC Profile Working Group -- and reflected in this document -- will be an evolutionary one [THEIMER].

To generate a precise set of set of requirements, this document first identifies (Section 2) a simple base case will be defined that we expect to have universally implemented by all batch job scheduling clients and schedulers. It is important to recognize that the base case represents the simplest ***HPC*** use case, not the simplest ***HPC Grid*** use case. Assuming that the term "Grid" refers to interactions among parties spanning organization boundaries, this use case reflects the fact that there are important and common HPC use cases that do *not* span multiple organizations. (Note: for a glossary of terms for OGSA, see [TREADWELL]).

Section 3 identifies a set of important use cases that any HPC interoperability design should be able to cover as *selectively* offered extensions to a simple, basic, HPC interoperability design. We refer to these as ***common cases***. Common use cases represent use cases that some significant fraction of the HPC community wishes to support. They do not represent use cases that *all* participants of the HPC community are expected to support. For example, it is conceivable that some significant fraction of the HPC community will eventually participate in a global distributed "grid file system" that spans virtual organizations. However, it may be equally likely that a significant fraction – perhaps even a majority – of the HPC community does not participate in such a system.

The set of common use cases defined implicitly represent the class of use cases at which an extensible HPC Grid interoperability design will be targeted. It should be understood that esoteric use cases not included in the common use case set may not be supportable by the extensible HPC Grid design that this design effort (and eventually the associated working group) will be creating. That is, to enable better understanding and predictability of the implications of composing multiple, independent design extensions a limited form of extensibility will likely be chosen rather than a fully general one. How limited a form of extensibility is chosen will be informed by the requirement that the set of common use cases defined must be realizable by it.

Use cases capture client-visible functionality requirements rather than being technology/system-design-driven. Thus, for example, a use case might specify that a "meta-scheduling" service be available that allows users to submit jobs to the set of all compute clusters within an organization. In this document the use case would not concern itself with how that meta-scheduling service is implemented or what component services it might consist of.

2. Base Case

The simplest HPC use case is a high-throughput compute cluster that is managed by a batch job scheduler and that is used only from within an organization. In this section, we cover the user interface, the state diagram, resource descriptions, fault tolerance model, out-of-band considerations, and out-of-scope considerations.

2.1 User Interface

Users can make the following requests to a job scheduler:

- *Submit* a job for execution, with a specification of resource requirements. The scheduler will return either a scheduler-relative unique jobID or a fault reply. Once a job has been submitted it can be cancelled (see below), but its resource requests can't be modified.
- *Query* a specific job for its current state. Users can only query jobs that were submitted using their user credentials. The scheduler will return either a standard description (to be defined) of the specified job's state, including its current execution state (see below) and a copy of the submission request, or a fault reply if the job is unknown.
- *Cancel* a specific job. Users can only cancel jobs that were submitted using their user credentials. The scheduler will return either an indication that the cancellation succeeded or a fault reply.
- *List* jobs. Users can request a list of the IDs of all jobs that have been submitted using their user credentials. The scheduler will return either a list of jobIDs or a fault reply.

2.2 State Diagram

The state diagram for jobs that have been submitted by a client to a job scheduler contains three states: *queued*, *running*, and *finished*. A job is in the *queued* state while it is waiting to run. Once the scheduler has dispatched the job to actually run on resources that have been allocated to it, the job is in state *running*. When the job terminates – for whatever reason – it enters the *finished* state. Jobs may terminate because they successfully exit, exit with an error code, get cancelled by someone, or because the resources they depend on fail in some manner.

Jobs eventually time out and are garbage collected if they don't finish of their own accord or get cancelled by a user request. When a job exits (for whatever reason) information is kept about the following:

- Whether it exited successfully, with an error code, or terminated due to some other fault situation.
- How long it ran in terms of wall-clock time.

This information is kept for a time period determined by the administration policy of the compute cluster.

2.3 Resource Descriptions

Only a small set of "standard" resources can be specified in resource descriptions, such as number of CPUs/compute nodes needed, memory requirements, disk requirements, etc. While not described in this document, it is assumed that there are (or at least will be) standardized, well-known semantics associated with the meanings of these resource descriptions and any associated scheduling behaviors they imply. Part of the semantics that must be defined for these resource descriptions is the set of comparison operations that may be used in a description. Only equality of string values and numeric relationships among pairs of numeric values are provided in the base use case.

2.4 Fault Tolerance Model

If a job fails due to system problems then it must be resubmitted by the client (as a new job); the job scheduler will not automatically rerun the job.

The scheduler can fail but must keep information about jobs it has received on persistent storage. The scheduler may discard information about jobs that have finished longer than a given time period ago, with the value for that time period being a standardized value (likely to be at least 1 day). If persistent storage fails then all job information – past and present – is assumed lost.

Failure of the scheduler may or may not cause currently running jobs to fail.

2.5 Out-of-Band Aspects

Data access issues are considered to be out-of-band, e.g. because of the presence of a distributed file system. Stated somewhat differently, data access is considered to be an application issue rather than something that is supported as part of the underlying HPC infrastructure (for this base use case).

Program provisioning is also considered to be out-of-band. That is, programs are assumed to be pre-installed – either directly on the compute nodes of the cluster or on something like a distributed file system that is accessible from the nodes of the compute cluster.

Creation and management of user security credentials is considered to be out-of-band. Users are assumed to have the necessary credentials needed to submit both data and work to the compute cluster. Note that this covers, among other scenarios, systems where all jobs are run using “anonymous” credentials and users must explicitly stage any private data they wish to use to common “pool” storage facilities.

The compute cluster's job scheduler is reachable at a well-known communications endpoint. Thus there is no need for directory services beyond something like DNS.

Management of the system resources (i.e. compute nodes) and services of the compute cluster is out-of-band and indeed opaque to users of the cluster. This is not to say that there isn’t a standard means of managing a compute cluster; just that system management is not part of the HPC cluster *use* case.

2.6 Out-of-Scope Considerations

Which scheduling policies a scheduler supports is out-of-scope. Similarly, concepts such as quotas and other forms of SLAs are out-of-scope.

Jobs are completely independent of each other. The notion of dependencies among jobs is out-of-scope.

A job consists of a single program running on a single compute node; infrastructure support for parallel, distributed programs – such as MPI programs – is not assumed in the base HPC use case.

Reservation of resources separate from allocation to a running job is out-of-scope. That is, things like reserving resources for use at some particular time in the future is not supported.

Interactive access to running jobs is out-of-scope. It is not prohibited, but is viewed as being an application-level concept that has no system support (at least in the base use case).

3. Common Cases

It is worth reemphasizing here that the purpose of enumerating common use cases is primarily to inform the design of the extension mechanisms that the HPC Grid profile working group will define. Various common use cases may be alternatives to each other while others may represent incremental extensions of each other.

3.1 Exposing existing schedulers’ functionality

Existing popular HPC batch job schedulers represent, almost by definition, common HPC use cases. A list of schedulers – presented in alphabetical order and not intended to be comprehensive by any means – to be explored to understand the functionality they offer includes

the following: Condor, Globus, LSF, Maui, Microsoft-CCS, PBS, SGE. The absence of an existing scheduler from this list should in no way be interpreted as being exclusionary.

3.2 Polling vs. Notifications

The base use case requires that client users and/or their software poll a job scheduler in order to determine the status of a submitted job. An alternative approach is to allow clients to optionally request the receipt of notification "call-back" messages for significant changes in the state of a job (meaning changes from *queued* to *running* or *running* to *finished*).

The base case does not include this capability because there is a spectrum of different delivery guarantees that can be associated with a notification interface. The easiest-to-provide design offers best-effort delivery semantics, where there are no guarantees concerning whether or how soon after a job state change takes place a notification message will be delivered to a registered client user. That is, fault events occurring within the job scheduler, the compute cluster, the network between the scheduler and the client, or elsewhere may cause notification messages to be delayed for arbitrary amounts of time or not delivered at all.

Unfortunately, whereas best-effort notification delivery semantics may provide significant efficiency gains over a polling interface, they do not reduce the programming complexity required to implement client software that may be trying to do something like orchestrate a workflow of multiple jobs. However, to support a significant reduction in client complexity would require substantially more difficult-to-provide guarantees of a system's infrastructure than best-effort notification delivery guarantees do. Since job schedulers are not required to be highly available in the base use case, the client must be prepared to poll in any case for the situation where a scheduler has crashed and will not recover for a longer period of time. Since network partitions can occur, the client must be prepared to deal with arbitrary delays in notification delivery and with the ambiguity of distinguishing between scheduler crashes and network partitions. Similarly, the scheduler must keep trying to delivery notifications in the face of client inaccessibility – be it due to client crash/disappearance or network partition. How often should the scheduler try to deliver a notification message? How to trade-off additional overhead vs. timeliness of delivery does not have an obvious answer. (Consider that a high throughput computing facility may have many thousands of jobs queued up to run, all of which can queue up notification messages to be sent out during a network partition.) All these considerations make it unclear whether there is a practically useful common case that involves the use of stronger than best-effort notification delivery semantics.

3.3 At-Most-Once and Exactly-Once Submission Guarantees

The base use case allows the possibility that a client can't directly tell whether its job submission request has been successfully received by a job scheduler or not. This is because the request or the reply message may get lost or the scheduler may crash at an inopportune time. Since the scheduler is required to maintain a persistent list of jobs that it has received and started executing, clients can obtain at-most-once execution semantics (subject to the scheduler's persistent storage facilities not being destroyed) as follows:

- If the client receives a reply message for its job submission request then it knows that the scheduler has received the request and has or will eventually execute the job.
- If the client does not receive a reply message for a given job submission request then it must issue a list-jobs request to the scheduler and compare the list of jobIDs returned against its own list of previously returned jobIDs to see if there are jobs that are unaccounted for.

Some systems allow clients to include a client-supplied unique jobID in a job submission request in order to enable things like suppression semantics for duplicated job submissions. This enables somewhat simpler client middleware code to be written since clients always have a list of jobIDs that they can employ to query a job scheduler about whether a particular job submission has been received or not.

While this is a reasonable extension use case to consider, it is not part of the base use case for the following reasons:

- Not all schedulers provide it.
- It requires that clients be able to verifiably create globally unique IDs, which requires infrastructure that is not currently assumed by the base use case. In particular, verifiable generation of globally unique IDs – so that buggy or malicious clients can't interfere with others' job submissions – requires suitable trust authorities and an associated security infrastructure, which are not presumed in the base use case.

An alternative approach to simplifying client middleware is to require exactly-once job submission semantics. In this case, a client knows precisely, at the time of submission, whether a job submission request has been received and persistently recorded or not. Furthermore, it knows this without having to verifiably generate a globally unique ID. The disadvantage of this use case is that exactly-once submission semantics require heavy-weight mechanisms, such as distributed two-phased commits between client and job scheduler, for their implementation.

3.4 Types of Data Access

The base use case assumes that data staging is something that occurs at the application level and hence is out-of-band to the HPC support infrastructure. At least two additional common use cases can occur:

- The HPC infrastructure explicitly supports non-transparent staging of data between independent storage systems.
- The HPC infrastructure explicitly supports transparent data access within a virtual organization or across a federated set of organizations.

Several different forms of explicit data staging can be imagined:

- The simplest form is simply the ability to copy files and directories of files between independent storage systems. Within an organization that may be as simple as using a (recursive) file copy utility. Copying data between systems that reside in different organizations requires additional support to deal with matters such as fire walls in particular and security in general.
- Users may also want convenience/efficiency features, such as "rsync"-style differencing capabilities between two copies of a data set. In this use case, clients copy a data set to a second storage system and then subsequently "synchronize" the two copies on demand by copying only the differences that have occurred on either side to the other side (and flagging conflicting updates for out-of-band – e.g. human – resolution).

Transparent data access – typically in the form of a distributed file system – can also be embellished with various additional features:

- The simplest form allows transparent (remote) access to data that can be named using a distributed, shared name space. As with explicit data staging, this may be fairly trivial within an organization and may require additional, mostly security-related capabilities when enabled across multiple organizations.
- More sophisticated systems may also support caching and/or replication of data. However, from a user's point-of-view, such features would be mostly transparent, unless also used in support of scenarios such as disconnected/partitioned operation of systems.

3.5 Types of Programs to Install/Provision/Run

Another axis along which various use cases can be distinguished is the kinds of programs to install, provision and run. The base case posits programs that are pre-installed, so that the supporting HPC infrastructure does not have to concern itself with issues such as provisioning. However, users may have programs that require explicit installation of some form. The simplest form of explicit installation involves copying the relevant files comprising a program (and its associated support data) to a location that is accessible from the compute nodes on which it is to run. A more complicated form of installation involves explicit subsequent installation operations,

such as updating of OS registries, etc. In order for the underlying HPC infrastructure to support this kind of extended installation, the process must follow a standardized set of installation steps. An interesting variation on the standardized installation approach is to employ client-supplied virtual machine images as the standard form of executable unit. This has several advantages:

- Complicated installation of things like services can be performed by the client before-hand; the HPC infrastructure need only understand how to receive a virtual machine image from a client and then run it on a suitable compute node.
- Cleanup after a job has finished is also very simple: all relevant data and processes to clean up are contained within the virtual machine.

Unfortunately virtual machines also have some disadvantages: They are potentially much larger than the corresponding programs that a client might supply otherwise. They may also incur an unacceptable performance overhead, for example, if their IO emulation is substantially slower than the underlying raw hardware.

3.6 Multiple Organization Use Cases

Arguably the most common *Grid* HPC use case involves submission of HPC jobs to a compute facility that resides in a different organization than the submitting client user does. This adds several addition wrinkles to the basic HPC use case:

- Submission of jobs requires additional security support in order to deal with either federation of identity information or, more simply, with authenticating a “foreign” user and mapping their identity to relevant credential and authorization information within the organization of the computing facility.
- Data residing on storage facilities controlled by the client’s organization must be either copied to suitable storage facilities controlled by the computing facility’s organization or made access across organizational boundaries. (This is one of the data access use cases described earlier.)
- If users from foreign organizations are trusted less than “local” users then their programs may need to be run with additional sandbox functions in place to prevent them from doing harm to the facilities of the computing facility or its organization.

An extension of the above use case is one in which the data needed by a program is controlled by a different organization than the one that either the job-submitting client or the computing facility reside in. In this case access to the data – either for purposes of explicit copying or as part of transparent remote access – must involve the use of delegated credentials/access rights.

3.7 Extended Resource Descriptions

The base use case assumes a small, fixed set of “standard” resources that may be described/requested in a job submission. Various systems may have additional kinds of resources that a client might want to specify, such as GPUs. One can imagine common use cases that define additional “standard” resource types. Another approach is to allow arbitrary resource types whose semantics are not understood by the HPC infrastructure, which deals with them only as abstract entities whose names can be compared textually and whose associated values can be compared textually or numerically depending on their data type. It is important to understand that, whereas the “mechanical” aspects of an HPC infrastructure can mostly be built without having to know the semantics of these abstract resource types, their semantics must still be standardized and well-known at the level of the human beings using and programming the system. Both the descriptions of available computational resources and of client requests for reserving and using such resources must be specified in a manner that will cause the underlying HPC “matchmaking” infrastructure to do the right thing. This matchmaking approach is exemplified by systems such Condor’s class ads system.

Another place where extended descriptions can occur is in the accounting information returned for a job. The base case only returns how long a job ran in terms of wall-clock time. Other common use cases may involve returning additional information, such as memory, disk, and network resources employed/consumed.

3.8 Extended Client/System Administrator Operations

Various common use cases involve client user-specified operations that are not supported by the base case. For example, users may wish to modify the requirements for a job after it has already been submitted to the job scheduling service – e.g. to extend the maximum lifetime it should be allowed to exist before getting forcibly terminated and garbage-collected.
Another common use case involves efficient interaction with arrays of jobs rather than just a single job. For example, a client user may wish to cancel all jobs currently running under their credentials – which might involve the cancellation of many hundreds or even thousands of jobs. Ideally this could be done with a single cancel request rather than having to issue hundreds or thousands of separate, individual cancel requests with all the attendant overhead that might be involved.

Several common use cases involve additional job scheduling operations that are most likely issued by system administrators rather than client users. These include suspension/resumption of jobs and migration of jobs among compute nodes (assuming that a given computing facility supports such operations – which may or may not be the case). Although implementation of such system administrator-issued requests may involve use of a separate system management infrastructure and associated interaction interfaces, the HPC infrastructure still needs to be aware of what is going on and may wish to reflect/expose some or all of the associated state information back to the owners of the affected jobs. For example, a job scheduling service might wish to expose the fact that a supposedly running job has been suspended.

An important aspect of exposing such information is that it must be done in a fashion that will not confuse simple client software (and unsophisticated client users) that will likely have no understanding of the semantics of such extended state information. Thus, for example, if a job that was in state *running* is suspended then client software and users expecting to see the job in one of the states *queued*, *running*, or *finished* should still see that state displayed, except perhaps with an extension or annotation that indicates that the job is in the *suspended* sub-state of state *running*.

3.9 Extended Scheduling Policies

There are a plethora of commonly employed scheduling policies beyond the simple one of first-come-first-served with round-robin allocation. These include everything from shortest/smallest-job-first to weighted-fair-share scheduling. Other concepts that are commonly supported are multiple submission queues, job submission quotas, and various forms of SLAs, such as guarantees on how quickly a job will be scheduled to run.

3.10 Parallel Programs and Workflows of Programs

The base use case concerns itself only with single instances of programs (also sometimes referred to as serial programs). There are several common use cases that involve multiple programs:

- Parallel, distributed programs, such as MPI programs require that the HPC infrastructure be able to instantiate such programs across multiple compute nodes in a suitable manner, including provision of information that will allow the various program instances to find each other within the cluster.
- Programs may have execution dependencies on each other. These may be static dependencies, such as which tasks must complete before a given task may be started. They may also be dynamic dependencies, implying the need for some sort of workflow orchestration system.

Whereas the HPC infrastructure needs to be cognizant of how to deploy MPI programs – so that concurrent program instances can be co-scheduled at the same time and can informed of which compute nodes the other instances are residing on – that is not necessarily the case for

implementing either static or dynamic dependency criteria. These can be implemented as client-side applications, although an HPC infrastructure might choose to offer support for this kind of functionality in order to enable a more efficient implementation thereof – e.g. by trying to schedule dependent jobs that employ the same data on the same compute node.

3.11 Advanced Reservations and Interactive Use Cases

One common use case is the ability to reserve resources for use at a specific future time, for example, to run an interactive application.

Support for interactive applications also requires that these applications be able to communicate in real time with external client users and/or their externally running software. Depending on the system configuration and security policies of the computing facility this may or may not be easy to support.

3.12 Cycle Scavenging

The base HPC use case involves submitting batch jobs to a dedicated compute cluster. A common variation on this scenario is cycle scavenging, where a batch job scheduler dispatches jobs to machines that have dynamically indicated to it that they are currently available for running guest jobs.

From a users' point-of-view a cycle scavenging infrastructure can be viewed as a "virtual cluster" whose compute nodes are potentially more flakey than those of a dedicated compute cluster (assuming that a cycle scavenging machine will terminate guest jobs whenever the machine's owner reclaims the machine for his own work). From the point-of-view of the owners of the donating machines the cycle scavenging infrastructure can be viewed as a "virus submission infrastructure" since foreign jobs might be buggy or even malicious. Thus, a key aspect of cycle scavenging use cases is whether or not they provide robust sandboxing guarantees to the owners of the machines being employed.

3.13 Multiple Schedulers

In an environment where there is more than just one computing facility, a common use case is to be able to submit work to the whole of the computing infrastructure without having to manually select which facility to submit to. Assuming that any compute clusters within the infrastructure still run their own job scheduling services, this use case involves employing *meta-schedulers* that schedule jobs by either dispatching them directly to individual computing facilities – e.g. to desktop machines that have made themselves available for cycle scavenging – or by forwarding them to the scheduling services that control some fraction of the computing infrastructure, be it a single compute cluster or a swath of multiple clusters and/or individual computing nodes.
The simplest meta-scheduler use case is a strictly hierarchical two-level one, in which there are only dedicated compute clusters and a single meta-scheduler that fronts them. All job submissions go to the meta-scheduler, who therefore has complete knowledge and control over which jobs should be scheduled on which compute clusters and forwards them to compute cluster job scheduling services accordingly.

A somewhat more complicated variation on this use case includes cycle scavenging nodes in the infrastructure. The meta-scheduler still receives all HPC job submissions that clients wish to execute remotely, but must be prepared to deal with nodes that may preempt running jobs to favor their owners' work and that may become altogether unavailable for arbitrary lengths of time. Both clients and the meta-scheduler must also understand the security implications of running various jobs either on dedicated compute clusters or on arbitrary cycle scavenging machines.
Another variation is to create an arbitrary hierarchy of meta-schedulers. For example, if the computing infrastructure is spread across the wider area or across multiple organizations then it may make sense to have a meta-scheduler per geographic site and/or per organization.

Strictly hierarchical – and hence inherently centralized – meta-scheduling schemes have the advantage that they can avoid various problems inherent in decentralized scheduling environments wherein imperfect scheduling information can lead to quite poor scheduling decisions under various circumstances. However, such schemes also have significant disadvantages, including the danger of bottlenecks and single points of failure (and/or partitioning of top-level meta-schedulers away from clients and segments of the infrastructure that could service local clients).

An alternative is to present users with an environment in which there are effectively multiple competing scheduler services. One of the simplest use cases is one in which users submit their jobs directly to the job scheduling service controlling a local "preferred" compute cluster. That scheduler either runs the job locally – perhaps even preferring it over queued "remote" jobs – or forwards it to a meta-scheduler for execution somewhere in the rest of the wider computing infrastructure. From the users' point-of-view this may look effectively the same as a strictly hierarchical meta-scheduling design, except that it may result in a greater variation in job service times: jobs that can get run "locally" will likely have noticeably quicker completion times (perhaps also because they don't have to traverse the wider area to be submitted to a remote meta-scheduler) whereas those jobs that end up getting executed "remotely" will suffer noticeably longer completion times because the meta-scheduling system is effectively now scavenging resources from locally controlled compute clusters.

Once the notion of competing schedulers has been introduced one can imagine scenarios in which multiple competing meta-schedulers offer different kinds of scheduling policies. From the users' point-of-view such an environment may offer greater variety in available scheduling policies, but it may also yield poorer overall utilization of resources, depending on how competing schedulers partition or share those resources among themselves.

4. Security Considerations

The use cases in this document all rely on a security infrastructure based on a Web services platform, most notably the WS-I Basic Security Profile Version 1.0 [BASIC].

5. Contributors

This document greatly benefited with the dialogue and comments of members of the OGF Working Groups (including the HPC Profile WG, the BES WG, and the JSDL WG). Specific individuals to which the authors acknowledge in the development of these ideas include: Glenn Wasson, Ian Foster, Andrew Grimshaw, Carl Kesselman, Karl Czajkowski, and Dave Snelling. We apologize for any contributors we have inadvertently omitted.

6. Intellectual Property Statement

7. Disclaimer

This document and the information contained herein is provided on an “As Is” basis and the OGF disclaims all warranties, express or implied, including but not limited to any warranty that the use of the information herein will not infringe any rights or any implied warranties of merchantability or fitness for a particular purpose.

8. Full Copyright Notice

9. References

[BASIC] Barbir, A., M. Gudgin, M. McIntosh, K.S. Morrison, eds. Basic Security Profile Version 1.0. Web Services Interoperability Organization (WS-I) Working Group Draft. 2006-03-29.

[THEIMER] Theimer, M., S. Parastatidis, T. Hey, M. Humphrey, and G. Fox. An Evolutionary Approach to Realizing the Grid Vision. Feb 13, 2006.

[TREADWELL] Treadwell, J. ed. Open Grid Services Architecture: Glossary of Terms. OGF GWD-I. Jan 25, 2005.