

Condor services for the Global Grid: Interoperability between Condor and OGSA *

Clovis Chapman¹, Paul Wilson², Todd Tannenbaum³, Matthew Farrellee³,
Miron Livny³, John Brodholt², and Wolfgang Emmerich¹

¹ Dept. of Computer Science, University College London,
Gower St, London WC1E 6BT, United Kingdom

² Dept. of Earth Sciences, University College London
Gower St, London WC1E 6BT, United Kingdom

³ Computer Sciences Department, University of Wisconsin
1210 W. Dayton St., Madison, WI 53706-1685, U.S.A.

Abstract

In order for existing grid middleware to remain viable it is important to investigate their potential for integration with emerging grid standards and architectural schemes. The Open Grid Services Architecture (OGSA), developed by the Globus Alliance and based on standard XML-based web services technology, was the first attempt to identify the architectural components required to migrate towards standardized global grid service delivery. This paper presents an investigation into the integration of *Condor*, a widely adopted and sophisticated *high-throughput computing* software package, and OGSA; with the aim of bringing Condor in line with advances in Grid computing and provide the Grid community with a mature suite of high-throughput computing job and resource management services. This report identifies mappings between elements of the OGSA and Condor infrastructures, potential areas of conflict, and defines a set of complementary architectural options by which individual Condor services can be exposed as OGSA Grid services, in order to achieve a **seamless integration of Condor resources in a standardized grid environment**.

1. Introduction

The definition of the Open Grid Service Architecture (OGSA) as a development standard was an important step towards creating a seamless grid infrastructure encapsulating services and resources worldwide. Though it is currently in the process of being re-factored into a family of standards, the Web Service Resource Framework (WSRF) and Web Service notification (WS-notification), the concepts it defines remain: by building on XML-based Web Service standards for communication and service description, and incorporating mechanisms enabling the creation, naming and discovering of stateful transient Grid service instances, OGSA aimed to provide an extensible, manageable and dynamic framework to support the global grid.

For such standards to be embraced by the Grid community it is important to investigate the development potential for standard compliant versions of existing grid tools and middleware. Condor is an example of a widely adopted software package firmly established within current Grid computing. It is a feature-rich package providing high-throughput computing resource and workload management

software for cycle scavenging on heterogeneous distributed systems. However whether or not it remains at the forefront of Grid computing will largely depend on its ability to adapt to emerging standards without affecting the features that make it so popular. The wide range of functionality provided by Condor, and the well-defined decomposition of its services, make it a prime candidate for such an investigation.

Bringing together Condor and Grid Service standards will not only further leverage acceptance of grid standards, but can also add significant new functionality designed to push Condor's boundaries. By embedding Grid considerations within the Condor architecture we considerably increase the potential for integration of its various services in a standardized Grid environment.

We present in this paper different architectural alternatives by which Condor services can be exposed as OGSA Grid Services. These alternatives will be evaluated with respect to current capabilities of Condor and other Grid technologies available.

* The research for this report was partially funded by NERC as part of the e-Minerals project.

2. Mapping the problem areas

2.1 The Open Grid Services Architecture (OGSA)

The *Open Grid Services Architecture (OGSA)*, delivered in July 2003, is in essence a marriage of *Grid* and *Web Service* technologies and concepts. Central to OGSA is the notion of Grid Services: “*Web Services providing a set of well-defined interfaces (service discovery, dynamic service invocation, lifetime management and notification) and that follow specific conventions (naming, upgradeability)*” [2].

There are very strong motivations for the Grid community to adopt Web Services: Web Services provide increased levels of manageability, extensibility and interoperability between loosely coupled services across heterogeneous environments. OGSA brings these advantages to the grid community by extending W3C Web Services standards such as the Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and WS-inspection to incorporate grid specific concepts and requirements, namely:

- *Naming*: A Grid Service must be uniquely identifiable. All Grid Service instances, whether transient or not, are identified by a globally unique and invariable *Grid Service Handle (GSH)*. This *logical* name comprises of a URI and a scheme for its resolution to a *Grid Service Reference (GSR)*, which contains instance-specific information such as network addresses, service definitions and network protocol bindings and has a limited lifetime. Mappings are maintained by a *handle resolver* service.
- *Transient services and service lifetime management*: Transient services maintain state specific to a requested set of activities, and means to dynamically create and terminate service instances when no longer required must be provided. The dynamic instantiation of a transient service instance can be requested from a suitable *Factory* service, and individual grid service instances provide soft-state lifetime management operations.
- *Service meta-data management*: A Grid Service must be open to inspection, exposing its characteristics and public attributes (*service data*). Service data is in essence an encapsulation of XML elements.
- *Service discovery*: enabling users to discover suitable services through service

registries and service groups. A *service group* is in essence a directory maintaining entries of service GSHs and accompanying descriptive service data elements.

- *Notification*: Providing means to exchange asynchronous notifications between services. A notification from a source service (*notification source*) is in the form of a service data ‘push’ to registered clients: clients (*notification sink*) subscribe for one or more service data elements to be forwarded to them whenever they are updated.

2.2 Condor

The Condor Project has performed research in distributed high-throughput computing for the past 18 years, and maintains the Condor High Throughput Computing resource and job management software originally designed to harness idle CPU cycles on heterogeneous pool of computers.

In essence a workload management system for compute intensive jobs, it provides means for users to submit jobs to a local scheduler and manage the remote execution of these jobs on suitably selected resources in a pool. Condor differs from traditional batch scheduling systems in that it does not require the underlying resources to be dedicated: Condor will match jobs (*matchmaking*) to suited machines in a pool according to job requirements and community, resource owner and workload distribution policies and may vacate or migrate jobs when a machine is required. Boasting features such as checkpointing (state of a remote job is regularly saved on the client machine), file transfer and I/O redirection (i.e. remote system calls performed by the job can be captured and performed on the client machine, hence ensuring that there is no need for a shared file system), and fair share priority management (users are guaranteed a fair share of the resources according to pre-assigned priorities), Condor proves to be a very complete and sophisticated package.

Architecture overview

Condor’s key activities - job-resource allocation, job startup and execution, and metadata collection and display – are kept separate, allowing compartmentalization of Condor into clearly defined components, distributed amongst submission site, central manager and execution site, as illustrated in figure 1:

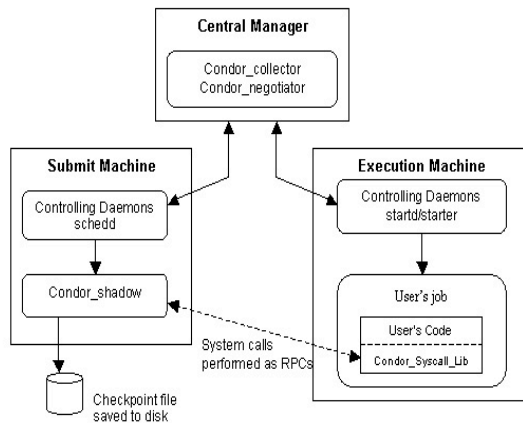


Figure 1: Condor Architecture overview

- **Central Manager:** For every condor pool a single central manager is responsible for collecting resource characteristics and usage information (i.e. accounting) from all machines in the pool and enforcing *community policies*. It is based on this collected information, and on *user priorities*, that job execution requests can be *matched* to suitable resources for execution during a negotiation cycle.
- **Submit Machine:** This system client allows users to submit jobs to a local virtual 'queue' (scheduler - *schedd*). The scheduler will request resource allocations for its jobs from the central manager during a negotiation cycle. Once a resource has been allocated to a job, the scheduler will spawn a *shadow* daemon responsible for managing the remote execution that job, and perform tasks such as state checkpointing, rescheduling the job in case of failure, or perform system calls made by the job running remotely on the local machine.
- **Execute Machine:** The execute machine, represented by the *startd* daemon, runs jobs on behalf of clients. It advertises its capabilities and usage information - as well as requirements and preferences upon a match - to the central manager, and manages the local execution of the job (via a spawned *starter* daemon), whilst protecting resource owner policies (e.g. a job may be vacated if the user touches the keyboard).

The service-based decomposition of the Condor architecture has enabled many of these services to be adapted for different uses and purposes, represented by different *Universes*, or Condor run-time environments. Apart from the *standard* universe - providing the entire set of Condor functionality, such as checkpointing and migration, to programs re-linked with a special

Condor library - and the *Vanilla Universe* - providing less features but suited to a wider range of programs, Condor supports specialized universes for *Java*, *PVM* (parallel applications), and *MPI* applications, as well as interaction with Grid resources managed by an array of grid middleware technology including the Globus Toolkit 3.x, Unicore, and others.

Class Advertisements

An important characteristic of Condor, central to its matchmaking capabilities, is its use of Class Advertisements (*ClassAds*). Matchmaking is a symmetric process; both job and machine requirements and ranks are considered when these are paired up. A *ClassAd* is a set of uniquely named expressions, using a schema-free semi-structured model. *ClassAds* enable mappings between attributes and expressions to be specified and evaluated with respect to another *ClassAd*. A *ClassAd* in Condor will either express a job's characteristics, requirements and preferences (e.g. memory, OS, etc.) or express the characteristics of a computing resource and any requirements or preferences upon the jobs it is willing to service.

3. Related Research

3.1 The Globus Toolkit 3 and the GRAM service

In order to leverage adoption of OGSA, the third incarnation of the Globus Toolkit (GT3), developed by the Globus alliance and released in June 2003, provided the first, Java-based, implementation of the Open Grid Services Infrastructure (OGSI), which included Grid Service containers supporting both stand-alone operation or deployment within J2EE Web or EJB hosting environments. It is this implementation that we use here as the reference implementation of OGSI.

The GT3 also brings in line with OGSA a number of higher-level services provided by its predecessor, such as the Globus Resource Allocation Manager (GRAM). The Globus GRAM is intended to provide a standard interface to job submission and monitoring for various underlying resource managers (Condor included). The GT3 GRAM defines the concept of a transient *Managed Job Service*, generated for each job submission, as a service abstraction over the underlying job scheduling process.

As a standard interface to multiple underlying schedulers, an obvious drawback is

that it is not possible for the GRAM to provide the complete set of functionality of every underlying system it supports. Defining a Condor-specific interface to the Grid, and having interface considerations embedded within the Condor architecture enables us to refine this interface to address the limitations that may be encountered when accessing Condor through the GRAM. In this way we do not have to limit ourselves to “wrapping up” an entire Condor pool as a job execution service but can also attempt to exploit the individual capabilities of Condor services in a Grid setting.

3.2 Condor and the Grid

The capability of managing jobs in an inter-domain setting, across independently managed resources, has been explored and introduced in Condor via the following mechanisms:

- *Flocking*: If a job cannot be serviced by resources of the local pool, the Condor scheduler can be configured to submit the job execution request to the central manager of another.
- *Condor-G*: Condor-G enables the scheduler daemon to submit jobs to resources presenting a Globus GRAM interface; providing job management services on the client side.
- *Condor glide-in*: Condor glide-in is a mechanism by which temporary resources managed by Globus can be added to a Condor pool. It enables Condor execution daemons to be submitted as jobs to local schedulers through the Globus GRAM interface. In effect, this mechanism enables users to build a personal Condor pool on resources independently allocated by different underlying scheduling systems. These Condor execution daemons will report back to a Condor collector when run on allocated resources, and Condor-G can then be used to submit jobs directly to these daemons.

We will attempt to demonstrate here means to consolidate and further improve on these concepts using OGSA.

4. Architectural Options

In this section we explore the different alternatives by which Condor services can be harnessed in a Grid environment by integrating OGSA mechanisms and concepts within the core of the Condor framework.

We structure the architectural options according to two models of job management and execution:

- *Delegation of job management responsibilities to local schedulers*: Exposing Condor services through OGSA enables us to support a model by which remote clients can request that the execution of a job on a pool of Condor resources be managed on their behalf by a scheduler local to that pool.
- *Controlled access to Condor-managed resources*: By integrating OGSA within the Condor framework, we can provide secure and controlled access to individual Condor managed resources, either directly (e.g. Computing on Demand), or through a well-defined allocation service.

These two complementary approaches cater for very different client requirements. Through delegations of job management responsibilities to local schedulers, remote clients are freed from the burden of coordinating resource usage and allocation. By using the Open Grid Services Architecture (OGSA) as a standard for defining and exposing Grid Services, we can identify, based on Condor's current architecture, a set of services through which remote clients can estimate underlying resource availability, submit jobs with corresponding parameters, binary input files and executables, and monitor the actual job execution.

However, the delegation process effectively shields these remote clients from any understanding of the underlying allocation process; remote clients must bind jobs to a pool queue using only estimates as to when a resource will be allocated to service that job. Whilst exposing descriptions of resource usage and pool policies may enable meta-schedulers to make more accurate predictions about the availability of pool resources, these clients would be considerably more efficient if provided with an interface to allocation services granting *controlled* access to managed resources: clients would then only bind a job to a resources when the resource has *actually been* allocated. Such a client, however, must be capable of dealing with the highly dynamic nature of the pool and the temporal availability of resources.

The primary motivation behind exposing services provided by Condor-managed resources is hence to enable scheduling and resource usage to be performed *across multiple independently allocated resources*; spreading

the entire set of Condor job management functionality, such as checkpointing and migration, across sites whilst respecting local policies and site autonomy. Exposing resource services also enables us to provide *Computing on Demand* functionality - special job requirements may necessitate that users request services *directly* from specific resources – as well as provide access to server side daemons in a Condor *glide-in* setting.

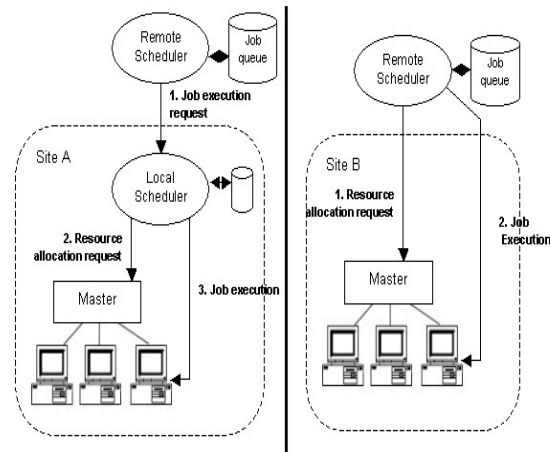


Figure 2: Delegation of job execution requests vs. direct access

4.1 Delegation of job management responsibilities to local schedulers

Supporting this model of job submission requires providing remote access to the following areas of functionality:

- *Job submission and queue management* - providing means for remote clients to submit job descriptions and accompanying input files and executables, as well as means of managing their jobs once submitted.
- *Job execution management* - including job monitoring and execution control.
- *Resource information providers* - which provide various descriptions of a set of resources, such as the underlying availability and type of resources or pool policies, enabling users to estimate the potential usage they can obtain from a set of resources before submitting jobs.

The transient scheduler

The Condor scheduler is in essence a customer agent responsible for managing a user's jobs. The scheduler should present remote clients with a transaction-oriented interface, complete with two-phase commit capabilities, for job submission and queue

management functions. The scheduler must also provide means to transfer files to and from the submission machine by incorporating specialized file transfer mechanisms.

Whilst the scheduler could be exposed as a *persistent* Grid service, we can gain many benefits from exploiting the *transient service* concept introduced by OGSA. Allowing users to generate one or more instances of the scheduler would enable the isolation of user-specific sets of related job and queue management activities. The notions of job scheduling and resource allocation in Condor are relatively distinct operations; resources are allocated on a fine-grained basis, based on user priorities and job pre-emption rules and this would be further highlighted by not obligating users to share a single scheduler instance at the submission site. The scheduler instance can then be cleanly destroyed upon termination of the requested activities.

We also envisage considerable security benefits: Whilst a Condor scheduler does not have to be run with root access, it is currently very much preferable, particularly when multiple users share a submission site. By allowing users to instantiate their own instances of the scheduler, we effectively eliminate the need for the scheduler to have root access. The responsibility of managing that privilege would be relegated instead to the scheduler factory.

Job representation

Adopting OGSA mechanisms allows us to explore different job representation strategies: how a job is perceived and how it is accessed and managed externally can and should vary during its lifetime in a Condor pool.

As previously described, Condor relies on *job ClassAds* to specify parameters and requirements for a job, and to provide a representation of a job and its status during its lifetime. By exposing the scheduler as a Grid Service, the representation of a job at submission time would henceforth be as a parameter of a scheduler invocation, essentially a data structure - in the form of a *job ClassAd* - containing all the parameters required to create the execution environment.

However once allocated to a resource, considerable benefits may be gained from providing a service abstraction of a running job - encapsulating state and execution management functions of a specific job, as well as more complex 'intra-job' management functions. This role could be taken on by the *shadow*, spawned as a transient grid service with lifetime

management functions tied to the lifetime of a job. Though the scheduler itself could be a shadow factory, with job requests hence embedded within service instantiation requests, a much preferable approach would be to have the creation of a shadow managed by a separate factory service on the same or different host. This would enable us to deal with the synchronous implementation of factories in OGSi implementations: the client does not have to be blocked until the job is allocated to a resource, as the scheduler will itself perform the service instantiation only when required.

Exposing queue contents and asynchronous notifications

Whilst an interface to the query API of the scheduler could be provided - through which external clients can obtain information about the queue - queue contents should be exposed as a collection of *service data* elements. Exposing *job ClassAds* as individual service data elements enables external applications to query the scheduler about job queue information relying on OGSA service data query mechanisms.

This also enables us to notify registered clients of job state changes. The asynchronous notification mechanism and *service data* are very much interlinked - a notification being a 'push' of service data to subscribed clients. We envisage that upon submission of a job request, clients will be subscribed with the scheduler service in order to receive information about job state changes in the form of updated *job ClassAds*. A potential source of concern is the lack of reliability of asynchronous notifications in OGSA: notifications bound to service data imply a focus on content availability rather than event notification, which is why exposing *job ClassAds* as *service data* elements fits very much within OGSA's approach to notification.

The collector

The Condor collector gathers information about availability and type of resources in the form of resource *ClassAds*. Exposing the collector enables external applications to request information about the state of a pool and estimate resource availability before submitting a job to the scheduler. In order to enable external applications to query the collector using OGSA mechanisms, the collector should be exposed as a Grid service, with XML representations of *resource ClassAds* exposed as individual service data elements.

Information obtained from the collector could be complemented with exposing the policies by which resources will be allocated to users to service their job, as maintained by the negotiator. However such a service would require a more suitable representation of customer capabilities to be adopted in Condor, encapsulating not only priorities but also authorization and resource access control information.

4.2 Providing controlled access to Condor-managed resources

Whereas our first approach defined a set of external interfaces to the client services of a Condor pool, we now move on to illustrate how OGSA can be embedded within the Condor framework in order to provide controlled access to the Condor resources themselves.

In order to support this model, we must consider how resources can be requested and obtained through an *allocation* process - consisting of *matchmaking* and *resource claiming* - authorized by the central pool manager based on community policies; and how individual resources services can be exposed.

The central manager

The functionality of the central manager in Condor is provided in terms of Condor daemons by both the collector, which is responsible for collecting *ClassAds* sent by all daemons in a Condor pool - including the *startd* advertisements stating resource characteristics, availability and preferences and scheduler advertisements stating that a particular user has idle jobs - and the negotiator, who will, based on community policies, match during regular negotiation cycles the resource advertisements to the scheduler requirements and inform interested parties.

In this context, where resources themselves are exposed as grid services, the collector concept is not entirely dissimilar to that of an OGSA Service Group, or registry. The collector could be extended - or wrapped - to present an OGSA service group interface, and store *Grid Service Handles* along side XML representations of *ClassAds* describing further characteristics, enabling all services to register themselves.

Based on information in the collector, the negotiator will, during a negotiation cycle, contact in priority order schedulers holding jobs in order to obtain information about their resource requirements. It is based on this information that a resource can be matched to

this scheduler, and that both resource and scheduler can be informed of the match, which they are free to accept or refuse. The fact that resource requirements need to be obtained from the scheduler during a cycle implies that a client needs to be exposed a grid service in order to provide an interface through which this can be achieved. Similarly, the *startd* must present an interface through which it can be informed of the match.

Client considerations

In order for a meta-scheduler to take full advantage of such an environment, it must be capable of distinguishing allocation requests from job execution submissions. As a client can choose to bind a job to a resource only when it has actually been allocated the resource, it needs to be capable of managing allocation requests separately from job execution requests in order to take full advantage of this framework.

To a certain extent Condor already offers this capability. When using Condor glide-in, the *condor_glidein* program can be used to request the allocation of resources from a specific site and schedule the execution of server side daemons on the foreign management system.

Resource representation

Condor resource-side architecture maps quite well to OGSA's factory concept. The *startd* could be exposed as a *Factory service*. Upon activation at startup, it will publish its description with the collector (exposed as an OGSA registry) as well as present an interface through which the negotiator can inform it of a match. Once matched, the *startd* can spawn individual starter instances exposed as transient Grid Services through which remote clients can manage the local execution of jobs. It should be noted that the OGSI GT3 implementation provides a lazy instantiation scheme, by which, whilst a grid service handle is generated and returned by a factory when a service instantiation is received, the actual service instance is only generated and deployed when a client attempts to resolve the corresponding GSH. In short a service instance is not actually created until the first attempt to communicate with it.

However, communication between client and starter is not solely one-way, as the starter will communicate with the shadow on the client host to request system calls and forward checkpoint data, etc. Support for this functionality, will require exposure of the

shadow as a Grid Service instance, which ties in well with our previous description of how the shadow can be exposed, but with focus on 'internal' communication with the shadow and its remote system call (RPC) functionality.

We must also consider the value of such a representation in its own right and not solely with regards to the above resource management services, as the actual use of a Factory also allows us to support Computing-On-Demand. Clients could, if authorized to do so, request job execution services directly, or support the use of Condor glide-in: communicating with these components using standardized OGSI (e.g. notification, service data querying, etc.) and GSI (e.g. credential mappings) communication and invocation protocols can add an extra layer of control and security over their use (e.g. for communication through firewalls). In the specific case of Condor glide-in the daemons to be run by a foreign resource management system could be submitted with a self-contained OGSI hosting environment.

4.3 Security and Identity management

We must adopt a two-layered approach to authorization and access control: whilst a security infrastructure such as the Grid security Infrastructure (GSI3 – part of the GT3) can provide us with remote user authentication and credential mappings between global identities (Grid certificates) and identities local to the Condor pool, actual authorization and access control should continue to be performed by Condor based on these local identities. The Condor architecture defines an access control and authorization framework tailored to its needs, specifying different roles (administrator/owner/negotiator/user) and levels of access (read/write) to components. Another important particularity of Condor is that it only requires users to have an account on the submission machine of a pool.

Whereas a grid map-file, which maps global identities to local accounts, could be used for the submission services, this does not necessarily apply to other Condor services which actually require a modification to the mapping mechanisms in order to cater for the distributed operation of Condor. In effect, whilst a GSI map-file will allow administrators to specify mappings between a certificate's *Distinguished Name* (DN) and a *username*, a Condor specific mapping would require mappings from DNs to *username@domain* combinations. Condor components can currently be configured to use the previous version of GSI in this manner, and this

capability should be updated where needed to incorporate the OGSI capabilities of GSI 3.

4.4 Grid Service Container

Though the service container has been referred to on a number of occasions, we should briefly consider its actual role in the system, and potential mappings to functionality provided by the *condor_master*. The role of the master in Condor is to ensure that daemons that should be run on a particular type of machine are started, and monitored for failure, in which case the master will restart them. It also allows for administrative commands to be issued either locally or remotely, such as reconfiguring daemons or turning them off.

In an OGSA environment, this responsibility is left to the service container and hosting environment (e.g. J2EE server). The ability to deactivate and reactivate services, and the fact that we can define as part of the Grid service implementation the course of action to be taken when activating/deactivating a service, should allow master functionality to be performed via the container. For example, we could allow the configuration file to be reprocessed whenever a service is reactivated. However, the ability to access this functionality remotely is limited by the capabilities of the hosting environment. A J2EE server for example may allow remote management of Grid Services.

5. Evaluation and conclusion

This investigation aimed to provide a relatively complete overview of the different levels of interoperability between Condor and OGSA. However, though some elements may provide interesting future developments, in the immediate term, we cannot recommend that every daemon in Condor should be replaced by an equivalent Grid service. The focus should be on providing external access to Condor functionality, with minimal interference to the overall intricate relationships between Condor components (such as the shadow and starter).

An excellent starting point would hence be to provide external access to the scheduler, taking advantage of concepts such as service data and transient service instances to boost Condor capabilities in a Grid environment. The ability for Virtual Organizations to instantiate their own scheduler instances, coupled with external administrative services such as discovery and monitoring services, would provide the basis for a powerful set of VO-wide management tools, which will be the focus of our future implementation work.

Though the investigation was conducted from an OGSA perspective, implementation will be conducted using the superseding Web Service Resource Framework and WS-notification family of standards. It should be noted that this evolution in grid standardization does not reduce the worth of such an investigation. The concepts and functionality defined by OGSA will be left mostly unchanged: “*WSRF essentially retains all of the functional capabilities present in OGSI, while changing some of the syntax [...], and also adopting a different terminology in its presentation*” [8].

References

- [1] Chapman, C., Wilson, P., Emmerich, W., Tanenbaum, T., Farrelle, M., Livny, M. Condor Services for the Global Grid, draft report, National Environment Research Council, 2004, <http://www.cs.ucl.ac.uk/staff/c.chapman/ogsa-condor-draft-030304.pdf>
- [2] Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002.
- [3] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maguire, F., Sandholm, T., Vanderbilt, P., Snelling, D. Open Grid Services Infrastructure (OGSI) Version 1.0. *Global Grid Forum Draft Recommendation*, 2003.
- [4] Sandholm, S., Tuecke, S., Gawor, J., Seed, R., Maguire, T., Rofrano, J., Sylvester, S., Williams, M. Java OGSI Hosting Environment Design – A Portable Grid Service Container Framework. Globus Project, 2002
- [5] Condor Team, Condor Version 6.6.0 Manual. University of Wisconsin-Madison, 2003
- [6] Livny, M., Tannenbaum T., Thain, D. Condor and the Grid, in Fran Berman, Anthony J.G. Hey, Geoffrey Fox, editors, *Grid Computing: Making The Global Infrastructure a Reality*, John Wiley, 2003.
- [7] Butchart, B., Chapman, C., Emmerich, W. OGSA First Impressions: A Case Study using the Open Grid Service Architecture, in *Proc. Of the All Hands Meeting 2003*, Nottingham, 2003.
- [8] Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Sedukhin, I., Snelling, D., Storey, T., Vambenepe, W., Weerawarana, S. Modelling stateful resources with Web Services, Version 1.1, IBM, 2004