Condor services for the Global Grid†

An investigation into the development of Condor Grid services with OGSA

Clovis Chapman, Paul Wilson and Wolfgang Emmerich
University College London
Gower St, London WC1E 6BT
{c.chapman|p.wilson|w.emmerich}@cs.ucl.ac.uk

Todd Tanenbaum, Matt Farrellee and Miron Livny
University of Wisconsin
Madison, Wisconsin, USA
{tannenba|matt|miron}@cs.wisc.edu

Abstract

In recent years grid computing developments have accelerated across both industry and a wide range of academic research. However, the lack of a standard grid development architecture has so far barred the door to a truly manageable, extensible and integrated 21st century grid infrastructure. The Open Grid Services Architecture (OGSA), developed by the Globus Alliance and based on standard XML web services technology, is the first attempt to provide the architectural components required to migrate towards standardised global grid service delivery. In order for existing grid middleware to remain viable well into this next generation of the grid it is important to investigate their potential for integration with emerging grid standards and architectural schemes. This study presents different architectural alternatives for 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 technologies and cement its place in this emerging Grid infrastructure; as well as providing 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 an OGSA environment.

† The research for this report was partially funded by the National Environment Research Council as part of the E-Minerals project.
# Table of Contents

Executive Summary ............................................................................................................. 1

1. Motivation and introduction .......................................................................................... 3
   1.1 The Grid .................................................................................................................... 3
   1.2 The Open Grid Services Architecture and Condor ................................................. 3

2. Mapping the problem areas ............................................................................................ 6
   2.1 The Open Grid Services Architecture (OGSA) ......................................................... 6
      2.1.1 OGSA Overview ................................................................................................. 6
   2.1.2 The Open Grid Services Infrastructure ............................................................... 6
   2.1.3 The Globus Toolkit ............................................................................................... 8
   2.2 Condor .................................................................................................................... 8
      2.2.1 Condor Overview .............................................................................................. 8
   2.2.2 Condor ClassAds ............................................................................................... 10
   2.2.3 Condor Architecture .......................................................................................... 10
   2.2.4 Condor and the Grid ......................................................................................... 13
   2.3 Bringing together OGSA and Condor ..................................................................... 13

3. Architectural Options ...................................................................................................... 15
   3.1 Delegation of job management responsibilities to local schedulers ....................... 16
      3.1.1 The Globus Resource Allocation Manager ....................................................... 17
   3.1.2 The scheduler .................................................................................................... 18
   3.1.3 The shadow ....................................................................................................... 21
   3.1.4 The collector .................................................................................................... 22
   3.1.5 The negotiator ................................................................................................... 22
   3.2 Providing controlled access to Condor-managed resources ..................................... 23
      3.2.1 Flocking and Condor glide-in .......................................................................... 23
   3.2.2 Condor resource management services ............................................................. 24
   3.2.3 The central manager ....................................................................................... 25
   3.2.4 Client considerations ....................................................................................... 26
   3.2.5 Resource representation ..................................................................................... 26
   3.3 Security and Identity management .......................................................................... 28
   3.4 Grid Service Container ....................................................................................... 29

4. Evaluation and Conclusion ............................................................................................. 30
   4.1 Evaluation ............................................................................................................. 30
   4.2 Conclusion .......................................................................................................... 31

References ........................................................................................................................... 33
Executive Summary

Grid computing is in a current state of flux, with exciting developments based around the idea of Web and Grid services. In order to harness concepts central to this service-based architecture, web and grid middleware must converge towards a commercially and academically acceptable standard conventions and open architectures. Condor and the Globus Toolkit are two examples of widely adopted software packages firmly established within current Grid computing. Whether either remains at the forefront of Grid computing will largely depend on their abilities to adapt to emerging standards without affecting the features that make them so popular.

Condor is a feature-rich package providing high-throughput computing resource and workload management software for cycle scavenging on heterogeneous distributed systems. One of Condor’s most important features is its ability to integrate expediently, reliably, securely and ‘out of the box’ with all major platforms and operating systems.

The Globus toolkit first appeared in 1996 and underwent three major evolutionary changes before emerging as the most widely utilized Grid middleware. The Globus Alliance collaborates with industry and the global Grid community to develop evolving grid-services standards, the outcome of which is Open Grid Services Architecture (OGSA), delivered in July 2003. OGSA aims to provide an extensible, manageable and dynamic framework for the creation of Grid services based on XML web services.

The definition of OGSA as a development standard is an important first step towards creating a seamless Grid infrastructure encapsulating services and resources world-wide. However, for such a standard to be embraced by the Grid community it is important to investigate the development potential for OGSA-compliant versions of existing grid tools and middleware. Such an investigation should highlight problems and benefits, architectural issues, enhanced or missing functionality, develop an understanding of common features, and provide feedback for the evolving grid standards development process.

This report provides such an investigation for Condor, one of the oldest and most reliable grid middleware systems. We propose combining the rich functionality and robust component framework provided by Condor with OGSA standards for defining and exposing grid services. This report identifies different alternatives to port the Condor architecture to an OGSA framework by exposing individual Condor components as Grid services, following two overall driving strategies: Firstly, we focus on enhancing remote client usability by defining Grid service interfaces to job submission, queue management, remote execution control and resource information provider Grid services, exposing Condor components such as the scheduler, the collector, etc. Secondly, we identify means to harness the individual functionality of Condor resource management and job execution components in an OGSA environment by providing secure and controlled access to resources managed by Condor either directly, or via a well-defined allocation service.

It is clear that the lack of maturity of Grid standards means that these will evolve rapidly, and the recent announcement of the decomposition of OGSA into a
family of standards such as the Web Service Resource Framework and Web Service Notification comes as no surprise. Nonetheless, in order for Condor to move forward with any future migration of Grid standards it is important to investigate the validity of integrating with currently proposed standards. Without such work, development of future Grid standards cannot be regarded as embracing the entire Grid infrastructure.
1. Motivation and introduction

1.1 The Grid

The motivation to enable ubiquitous global computing on demand is not a new one. The last 30 years have witnessed the emergence of data-rich, geographically distributed, heterogeneous enterprise-level computing requiring prodigious levels of computing power, be it memory, storage or CPU cycles.

This is illustrated by accurate prophesies of computing such as the following:
- “The time-sharing computer system can unite a group of investors... one can conceive of such a facility as an... intellectual public utility” (F. Corbato and R. Fano, 1966)
- “We will perhaps see the spread of ‘computer utilities’, which like present electric and phone utilities, will service individual homes and offices across the country.” (L. Kleinrock, 1967)

Grid computing is the latest paradigm to enter our vocabulary, but the concepts that define it are not new. A software and hardware infrastructure providing a seamless source of computational power is analogous with electricity grids (power on demand), but the metaphor is not complete. The services and functionality provided by computational Grids are varied and dependent on the wide range of underlying resources, whether CPU sharing, data storage, scientific or business applications, etc, and based on a two-way communication. Electricity is simply consumed, whereas grid computing requires return of results, metadata, and other responses to queries and requests from a distributed group of human or virtual users.

Therefore, a computational grid requires a more formal definition. A reasonable ‘checklist’ determining what is a computational grid has been proposed [1]. According to Foster, any grid has to satisfy three statements:
- The grid coordinates resources that are not subject to centralized control.
- The grid uses standard, open, general-purpose protocols and interfaces.
- The grid delivers non-trivial qualities of service.

These three statements merely hint at the technical, geographical and architectural problems involved in deploying a seamless extendable, integratable global ‘grid of grids’. Additionally, it is reasonable to argue about definitions present in these statements. It is the second of these statements that causes the greatest problem. Standardization is currently the bugbear in grid computing development. Where should we start? Which standards should we adopt? How do we ensure a common playing field?

1.2 The Open Grid Services Architecture and Condor

The explosion of international grid development only accelerates the need for standardization. An obvious starting point is to examine the existing software commonly used in grid computing. Amongst the technologies adopted by the grid community, both the Globus Toolkit and Condor stand out as internationally recognized standards within their areas.
The open Globus Toolkit has been in widespread global use for over 7 years, defining a set of protocols for distributed resource negotiation, sharing, authentication and access. The toolkit, now in its third incarnation, has become a de-facto standard for building computational grids and provides a set of resource and data management tools, enabling virtual organizations - encapsulating resources and users across multiple organizational boundaries - to share compute and data resources in a secure and controlled manner.

The soon-to-be-open Condor software has been around since 1986, pre-dating the grid by many years, and is the practical solution for the creation and management of high-throughput computing resources, providing a robust, sophisticated and powerful job and resource management system capable of handling the scheduling, matchmaking and execution of huge queues of jobs on large distributed resources.

Whilst these two examples are in widespread use within grid developments, and provide very usable real-world functionality, there are still many challenges to face if we wish to move towards a production-level grid infrastructure.

The problem area discussed in this report stems from the recognized identification of a lack of standardization within grid developments. Globus toolkit 2, Condor and in fact any other grid tool or software do not comply with any form of grid standard, unlike many other computing developments - the World Wide Web (www), for instance complies with w3c standards, and the internet with the IETF.

Recently this has begun to change with the introduction of OGSA, developed by the Globus alliance and the GGF and delivered in July 2003.

The Open Grid Services Architecture, (OGSA), represents the evolution of standardized grid services development based on XML and Web Services technology, and is certainly the only practical, standardized format for future grid developments, with significant commercial backing. Its developers hope that there will soon be a fourth checklist statement to add to the definition of a grid- that ‘…for an entity to be part of the Grid it must implement OGSA InterGrid protocols…’ (Foster, 2002).

Whether this statement will become fact is still to be proven, but it is a fact within computing that standards make for robust, reliable long-lived ubiquitous computing resources such as the internet and www, which provide acceptable levels of quality of service.

Already, the Globus Alliance has taken the next step via Globus Toolkit 3, extending existing GT2 functionality to address grid specific requirements using OGSA specifications and service semantics.

By embracing the flexibility, natural acceptance of heterogeneity and potential for extensibility available through OGSA, it is possible to investigate the added functionality that OGSA can bring to existing grid tools such as Condor, or alternatively, where OGSA could be extended or enriched through investigation of it’s ability to provide such added functionality.

Condor is already a formidable tool, providing a level of functionality unmatched by any similar product. Its richness, usability and scalability stem from a long, systematic development process and a professional production level of testing, support and deployment. However, to continue as a leading force in long-term future grid developments, it must embrace emerging standards. Condor is already moving forward in this area through efforts to allow it’s internal communication to use SOAP protocols over http, authenticate using x.509
certificates and formatting it’s metadata in XML form. Thus, Condor and web-
services are already holding hands. The next step? Condor Grid Services.

We propose to demonstrate methods of extending this development path by
exposing selected Condor services as OGSA Grid Services, with the aim of
expanding the intergratability of Condor into the proposed OGSA-based grid of the
future, and prove that OGSA can add significant new functionality designed to
push Condor’s boundaries.

We investigate in this report different alternatives by which the Condor and
Open Grid Services architectures can be brought together. Considering the wide
range of functionality provided by Condor, and the well defined decomposition of
its services, we attempt to identify the components and services it would be
sensible to expose, determine the mappings that can be performed between
elements of the OGSA and Condor architectures, and define potential areas of
conflict.
2. Mapping the problem areas

2.1 The Open Grid Services Architecture (OGSA)

2.1.1 OGSA Overview

The Open Grid Services Architecture (OGSA) is a very new paradigm in grid computing, delivered in July 2003, designed to standardize global grid development using mechanisms to create extensible, manageable and above all dynamic, domain-independent discoverable services within virtual organizations. It is in essence a marriage of Grid and Web Service technologies and concepts. There are strong motivations for the Grid community to adopt Web Services: Web Services provide interoperability and increased levels of manageability and extensibility between loosely coupled services across heterogeneous environments through the use of XML-based communication protocols (e.g. Simple Object Access Protocol - SOAP) and service descriptions (e.g. Web Service Definition Language – WSDL). OGSA attempts to bring these benefits to the Grid community through the definition 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, upgradable)” [2].

This is achieved via extending W3C web-services standards such as SOAP, Web Services Description Language and WS-inspection to include grid specific concepts, namely:

- **Naming**: A Grid Service must be uniquely identifiable.
- **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.
- **Service meta-data management**: A Grid Service must be open to inspection, exposing its characteristics and public attributes.
- **Service discovery**: enabling users to discover suitable services through service registries and service groups
- **Notification**: Providing means to exchange asynchronous notifications between services

2.1.2 The Open Grid Services Infrastructure

The Open Grid Service Infrastructure (OGSI) represents the physical infrastructure required to build a standard set of mandatory Grid-Service features that satisfy the grid concepts described above. Defined by the GGF as “the standard for next-generation grid-services infrastructure”, it aims to ensure a fundamental level of interoperability amongst all grid services. The OGSI specification defines standard interfaces, schema and behavior that must be implemented by Grid Services to fit within the OGSA vision. The specified interfaces are the following [3]:
• **GridService**: The root interface, encapsulating the root behavior of the grid service model and implemented by all Grid Services. It allows users to query a service about available public attributes/characteristics, or ‘service data’. Service data is in essence an encapsulation of XML elements, detailing service characteristics (e.g. GSH, GSR, instantiating Factory - see below) or application specific information. It also provides means for soft state management of a transient service’s lifetime; service instances are maintained until they expire, or are explicitly destroyed. Authorized clients may request the update of the expiry time of a grid service, according to their needs and service configuration.

• **HandleResolver**: 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. This separation between logical and physical name allows for location and invocation transparency, and upgradeability. The HandleResolver provides mappings between GSHs and GSRs. How services register their handles with the resolver is not defined by the specification, but implementations may for example require every instantiated service to register themselves with a local HandleResolver service.

• **Factory**: The dynamic instantiation of a transient service instance with particular characteristics can be requested from a suitable Factory service. In response to the request, the Factory would return a service locator containing a GSH and/or a GSR representing the instantiated service.

• **NotificationSource / NotificationSink / NotificationSubscription**: These interfaces provide means for asynchronous notifications to take place between services. A notification sink can register interest in receiving particular messages from a notification source. Registration for notifications generates a subscription instance, providing means for clients to manage the lifetime of their subscriptions. A notification from a source is in the form of a service data ‘push’ to registered clients: clients subscribe for one or more service data elements to be forwarded to them whenever these are updated.

• **ServiceGroup / ServiceGroupRegistration / ServiceGroupEntry**: Service groups enable the discovery of services by maintaining information about a group of services. The relationship between services in a group is not defined, a service group can simply act as a service registry or define a more complex link between services (e.g. federated services). It is through a registration service that a new entry is added to the service group, individual service group entries consisting of a GSH and accompanying service data describing the service.

OGSA and OGSI are currently the only available grid-specific standards, and have been embraced upon inception by industry heavyweights and the research community. Though the OGSI specification will be decomposed into a family of
Web Service specifications - such as the *Web Service Resource Framework*, and *Web Service Notification* - the new Grid concepts that it introduces, such as interoperability in heterogeneous environments, transient service state, dynamically discoverable services, etc., will remain and only be further refined by the superseding specifications. Due to the fact that the OGSI specification has been finalized and that a number of implementations of this infrastructure are available, OGSI can provide us with a concrete starting point for the integration of these new concepts into existing grid middleware and tools, such as Condor.

2.1.3 The Globus Toolkit

The Globus Toolkit - developed by the Globus Alliance - defines a set of individual protocols and tools, in the area of resource management, data management, security, etc. enabling users to build computational grids and share resources across organizational boundaries, without compromising site autonomy. Its current incarnation, the Globus Toolkit 3 (GT3), released in June 2003, is an attempt to leverage OGSA as a Grid standard, by providing the first implementation of the Open Grid Services Infrastructure. The GT3 provides a Java-based implementation of the grid service specification, as well as OGSI Grid Service containers supporting stand-alone operation or deployment within J2EE Web or EJB hosting environments.

It also brings a number of higher-level services provided by its predecessor, the Globus Toolkit 2, in line with OGSA advancements, and the following are of particular interest to us:

- *The Globus Resource Allocation Manager (GRAM)*: which provides a standard interface for jobs (in the form of executable binaries), to be run on a range of remote computational processing software systems (such as Condor, Sun Grid Engine, PBS, etc.), and supports job submission, monitoring and termination on remote resources.

- *The Grid Security Infrastructure (GSI3)*: The Grid Security Infrastructure has been brought in line with an OGSA security model in order to provide secure authentication and communication using SOAP and Web Service security specifications. It is based on a public key infrastructure and X.509 certificates, and provides support for delegation, which enables remote servers to perform service invocations on behalf of the user, and credential mappings, which enable Grid certificates to be mapped to site-specific local configurations and authentication/authorization mechanisms.

It is the GT3 implementation of the Open Grid Services Infrastructure that we use in this report as the reference implementation of OGSI, particularly with respect to its Grid Service container framework and its reliance on GSI as the security infrastructure.

2.2 Condor

2.2.1 Condor Overview
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 networks of computers to create high-throughput computing resources. Since its conception, Condor has gained a level of maturity and sophistication that make it a widely popular system, used by hundreds of government, academic, and commercial organizations worldwide.

The Condor system, which can best be described as a distributed job scheduler and resource management system, provides means for users to submit jobs in the form of executable programs, and manages the execution of these jobs on suitably selected resources in a pool, based on job requirements and community, resource owner and workload distribution policies. In this manner Condor differs from traditional batch scheduling systems for compute intensive jobs, as it does not require the underlying resources to be dedicated: Condor will select machines on which to run jobs which are not currently in use according to an arbitrary policy dictated by the owner of the resource. This ensures that underused resources in a pool are properly harnessed and may vacate and migrate jobs when the machine is required, effectively protecting resource owner policies.

Whilst the range of services provided by Condor is wide and varied, we can simplify Condor’s rich functionality into the following three categories:

- **Job scheduling**: A condor scheduler is responsible managing job execution requests - consisting of input files, execution binaries, environment requirements, and other parameters - from multiple users and maintaining a persistent queue of jobs. Users may assign different priorities to their jobs, and specify workflow dependencies between jobs.

- **Resource management services**: A central manager is responsible for collecting resource characteristic and usage information from machines in a Condor pool. It is based on this collected information, and on user priorities, that job requests can be matched to suitable resources for execution.

- **Job execution management**: Condor manages remote execution of jobs, providing file transfer mechanisms enabling required files to be staged on the remote machine, checkpointing mechanisms enabling job state to be saved and subsequently resumed, and migration of jobs from one machine to another. Further more, Condor enables system calls performed by the application running remotely to be performed on the client machine, providing a degree of transparency with regards to the execution of the job.

The service-based decomposition of the Condor architecture (covered below) 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.
2.x, Globus Toolkit 3.x, Unicore, and others. Many of these features will be explored in more detail in later sections.

2.2.2 Condor ClassAds

Condor Class Advertisements (ClassAds) are central to Condor’s matchmaking capabilities. A ClassAd is a set of uniquely named expressions, using a schema-free semi-structured model. ClassAds enable a mapping between attributes and expressions to be specified and evaluated with respect to another ClassAd. A ClassAd in Condor expresses a job’s characteristics, as well as requirements (e.g. memory, OS, etc.) and preferences upon matching a ClassAd representing a computing resource. Symmetrically, ClassAds representing computing resources specify attributes about the resource, and any requirements or preferences upon a job it is willing to service. The ClassAd framework provides a very effective means for job ClassAds to be matched to suitable resource ClassAds in a distributed policy environment.

2.2.3 Condor Architecture

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. This daemon-based decomposition of the Condor architecture is distributed amongst submission site, central manager and execution site, as illustrated in figure 1. Whilst there can only be one central manager, there can be any number of submit and execute machines in a given Condor pool, and a same machine may hold multiple roles:

- **Central Manager**: A single central manager exists for every Condor pool. It is responsible for collecting information about the pool and its machines, and the matchmaking of job requirements to suitable machines based on community policies in the form of job pre-emption rules and user priorities. Matchmaking is a symmetric process; both job and machine requirements and ranks are considered when these are paired up. Two daemons reside here, the *negotiator* and the *collector*. The collector is, as its name indicates, responsible for collecting ClassAds from all other daemons in the system (such as schedulers, or resource-side components), enabling us to determine the state of all machines in a Condor pool. It is based on this information that the *negotiator* will, during a regular negotiation cycle, match schedulers with waiting resource requests to available resources in priority order. The negotiator also performs accounting for the pool and maintains and adapts user priorities according to current resource usage. A previously established match may be revoked if a higher priority user/job requires a particular resource.

- **Execute Machine**: The execute machine runs jobs on behalf of clients. It advertises its capabilities and usage information - as well as requirements and preferences upon a match - with the central manager. The daemon specific to the execute machine is called the *Startd*, and represents a resource on the condor pool. When a successful match has been established
by the negotiator, and the submit machine specified by the match has established contact with the Startd, it will spawn a starter daemon responsible for managing the local execution of the job. The starter spawns the actual job on the execute machine, setting up the execution environment and monitoring the job’s progress once it starts. The starter communicates with the shadow daemon running on the submission machine (described below), returns status information, cleans up the execute machine upon job completion and exits. Resource owners can choose to cleanly vacate running jobs in order to regain control over the machine.

- **Submit Machine:** This machine allows users to submit jobs to a local virtual ‘queue’, represented by the schedd daemon. In order to service queued jobs, it will advertise to the collector the number of idle jobs it has in the queue in order to be contacted by the negotiator of the central manager during the negotiation cycle. It is then that the scheduler will submit requirements for its jobs in priority order so that it can be allocated suitable resources. Once a match has been established, the schedd will spawn a shadow daemon responsible for managing the remote execution of the job. The shadow will interact with the starter daemon on the remote machine and monitor its execution. It will manage the checkpointing of the job, perform system calls on the local machine on behalf of the remote application, and reschedule the job in case of failure. The scheduler may attempt to re-use a matched resource for another job or hold it as a spare, until the match is relinquished.

Common to all types of machines, is the master daemon. This daemon will run on every machine in a Condor pool. It is responsible for ensuring that the daemons for the corresponding type of machine are running and restarts them in case of

---

**Figure 1:** Condor Architecture overview
failure. It is also through this daemon that administrative requests such as start/stop or reconfigure daemons can be serviced remotely.

Interactions between Condor daemons are illustrated in figure 2. Whilst many of these processes will be covered in more detail throughout the report, it is important to mention the clear distinction between *matchmaking* and *claiming*. The role of the central manager, or more specifically the negotiator is a suggestive role. Whilst the negotiator may recommend a match between client and server, it is left for the resource itself to approve the match, based on owner configurations. Claiming a resource is hence kept distinct from matchmaking. When a client and resource have been informed of a successful match, the client will then proceed to claim the resource, which after mutual authentication based on information provided by the negotiator, can be authorized or rejected. This distinction enables Condor to cope with the fact that the matchmaking process may have been performed on potentially outdated information, and protect resource owner policies expressed independently from the community policies specified by a pool administrator. Further more the claiming protocol may vary according to the nature of the resource and security protocols.

![Diagram of Condor interactions](image)

**Figure 2**: Interactions between Condor daemons (in bold)

We should also consider the role of these components when dealing with parallel applications and the *master worker* paradigm. Essentially this paradigm allows a central master process to coordinate the execution of individual tasks by worker processes on remote nodes, and Condor can act as the resource management platform on which such applications can be deployed. Whenever the master requires additional nodes with specific characteristics on which to run tasks, it will request these from Condor’s resource management framework, which will spawn a new worker on a node to whom the master can then submit tasks. The Condor shadow in this case acts as the resource management service for this type of application, through which masters can request resources. In turn the shadow will proceed to submit these requests to the local scheduler. This master worker paradigm is one that should be noted, as it will drive many of the considerations as to how Condor resource management services can be exploited in the Grid.
2.2.4 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:

- **(direct) Flocking**: Flocking allows jobs idling in the queue of a given Condor pool to be run on spare resources on a different Condor pool. If resources of the local pool cannot service a job, the scheduler can be configured to submit the job execution request to the central manager of another pool.

- **Condor-G**: Condor-G enables the scheduler daemon to submit jobs to resources managed by Globus and other middleware packages. The previously mentioned *Globus Resource and Allocation Management* (GRAM) architecture essentially defines a 2-tier scheduling architecture: meta-schedulers can submit their jobs to a wide range of individual local schedulers through a standard and secure interface. Condor-G enables the scheduling component of Condor to be used to manage the submission of jobs to resources on the Grid presenting the GRAM interface, by 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. Condor glide-in effectively exploits the daemon-based decomposition of services in Condor by enabling 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. This strategy allows a distinction to be made between the allocation process of local schedulers and the actual job execution, and enable higher level schedulers to bind a job to a resource only when its availability is determined. Further more, the entire set of functionality of Condor can now be exploited through this strategy across multiple independently managed resources. In effect, Condor glide-in results in an end-to-end solution: in order to present semantic guarantees to the end user, such as at-most-once, at-least-once, and once-and-only once, a distributed job execution model must provide end-to-end management.

2.3 Bringing together OGSA and Condor

The mechanisms listed above have enabled Condor functionality to be exploited across organizational boundaries, either directly, through flocking, or through interoperation with Globus technology. This ability to interact with Globus technology – or more specifically the Globus GRAM – also demonstrates an understanding of the fact that no single resource management system can be suitable to all types of resources and applications on the Grid.
However to incorporate OGSA standards into the Condor framework provides a new dimension by means of which Condor functionality can be harnessed on the Global Grid. OGSA standards enable us to consolidate the concepts explored above into a grid-specific standardized framework, but also further extend the functionality of Condor. In effect, it provides new means by which Condor can integrate and interact with other OGSA compliant applications and resource management systems on the Grid.
3. Architectural Options

In this section, we explore the different alternatives by means of 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 resource 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 access to server side daemons in a Condor *glide-in* setting.
In terms of how we expose services, the required distinction between the concepts of resource allocation and job scheduling and management, and the need for us to define how Condor’s resource management services can interface with OGSA Grid Services, leads as a result to the definition of a resource management infrastructure capable of interfacing with a wider range of Grid Services. The well-established nature of Condor and the principles by which it operates would make it a particularly desirable platform for developing and deploying Grid Services requiring CPU scavenging and other high-throughput computing mechanisms.

### 3.1 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. These descriptions will enable users to estimate the potential usage they can obtain from a set of resources before submitting jobs to local schedulers.
In this section, we investigate how we can expose Condor services using OGSA in order to provide this functionality to remote clients. The aim is to define secure external interfaces to selected components by embedding the Grid Service layer within the Condor architecture. This approach enables us to fully integrate OGSA mechanisms within the Condor framework, rather than define a set of external gateway services.

Based on Condor’s daemon-based architecture – as described in Section 2.2.3 - we can map these various functionality requirements to the following Condor services:

- The scheduler (condor_schedd): is responsible for maintaining a persistent queue of jobs submitted by users. Exposing this service would provide an interface to job submission and queue management functions and is a potential source of queue related notifications.
- The shadow (condor_shadow): acts as a resource manager for a specific job (or set of jobs) on the submission machine, and can potentially provide control over the remote execution of the job and access to job state related notifications.
- The collector (condor_collector): Gathers information about availability and type of resources and job requirements in the form of resource and job ClassAds. Exposing a query API to this component would enable external applications to obtain information about the current state of a pool.
- The negotiator (condor_negotiator) is responsible for matchmaking users and resources as well as managing, via the accountant component, user priorities. It can potentially provide information about pool policies such as user priorities and job preemption rules in order for external applications to identify the semantics of the service provided by a pool.

We are able to build on the efforts that have been made in ‘SOAPing’ Condor. Several Condor daemons have been modified already to expose some of their functionality through Web Service interfaces. To date, the query API of the collector - which provides information about the state of pool resources - and the job submission and queue management functions of the scheduler have been made Web Service compliant. Extending these interfaces to become OGSI or WS-RF compliant enables us to circumvent the limitations of vanilla Web Service conventions and exploit OGSA mechanisms such as asynchronous notifications, transient service instances and Grid authentication and authorization through the Grid Security Infrastructure.

3.1.1 The Globus Resource Allocation Manager

Delegation of job execution requests to local schedulers as a means to exploit resources in the global Grid is a model that has been adopted by a number of Grid technologies, and in particular the Globus Toolkit. Comparisons with the OGSI implementation of the Globus Resource Allocation Manager (GRAM) are inevitable. The Globus GRAM is intended to act as a standard interface to job submission and monitoring, ensuring that remote clients do not have to deal with the varying differences and complexities of underlying local schedulers and
resource management systems. Without delving too deep into the GRAM architecture, we briefly describe some of its essential characteristics.

The GRAM defines the concept of a transient Managed Job Service, generated upon submission of a specific job, as a service abstraction over the underlying job scheduling process. Job requests describing characteristics and requirements of a job are specified using the Resource Specification Language (RSL), and are submitted to a Managed Job Service Factory as part of a service instantiation request. The instantiated Managed Job Service will interact with the local scheduler and provide a standard interface for users to monitor and manage the execution of a job in the remote environment. In order to transfer standard input and output between the remote client and the local resource manager, this service will spawn individual File Stream Service instances to stream specified files to/from a URL. Security is provided through the Grid Security Infrastructure (GSI). To ensure that local policies are respected, a user’s X.509 certificate will be mapped to a pre-defined local account under which service and scheduler invocations will be performed.

As the GRAM is meant to be a standard interface to multiple resource management systems, an obvious drawback is that it is not possible for it to provide the complete set of functionality of every underlying system it supports. Furthermore, the lack of an end-to-end solution destroys semantic job run guarantees, such as at-most-once, at-least-once and once-and-only once. Though the GRAM supports a 2-phase submission, an underlying resource management system such as PBS might not, and the semantic guarantee provided through a 2-phase commit is lost.

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.

### 3.1.2 The scheduler

The Condor schedd is responsible for managing a user’s job execution requests and maintaining a persistent job queue. It is its capability as a customer agent responsible for managing a user’s jobs that we aim to highlight through the Grid Service interface that we define. Rather than simply providing a job submission interface, the scheduler should present remote clients with a transaction-oriented interface, complete with two-phase commit capabilities, for job submission and queue management functions.

Apart from providing external applications with more control over their actions in terms of reliability, a specialized transaction-oriented interface enables us to deal with the synchronous nature of service invocations in the OGSI infrastructure, which is very much at odds with the queue based nature of Condor’s scheduling system.

As previously mentioned the scheduler has been extended to incorporate a Web Service interface with a 2-phase commit approach to job submission and queue management. We investigate here how OGSA functionality can be used to improve upon the work that has been achieved in exposing the scheduler interface.
• The scheduler factory

We first introduce the notion of a scheduler factory. 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.

The motivation behind allowing users to generate one or more instances of the scheduler would be to isolate 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.

A transient scheduler service may also provide locality benefits. If a user wishes to spawn a large number of jobs on a specific site, an instantiation request could be submitted to a scheduler factory on the remote site, in order to have an “on-location” manager.

We also envisage considerable security benefits to be covered in a later section when dealing with security and identity management.

• 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 - and we consider below means by which the condor_shadow can be exposed as an abstraction over the scheduling and management process of a specific job.

By creating different job representations according to different states, we reinforce the perception of the scheduler as an agent responsible for scheduling resource allocation requests, and introduce the notion of the shadow as a job-specific management agent.

We must also consider the means to transfer files to and from the submission machine. Condor currently provides an option to accompany a job ClassAd with a file ‘sandbox’ containing binary input and executable files and means to recreate this option within an OGSA environment must be considered. Unfortunately, the issue of data representation and virtualization in OGSA is very much a work in progress, and until standard data services built on OGSI are defined, we must contend with implementing or incorporating specialized file transfer mechanisms.
Exposing queue contents

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. Service data, essentially a structured set of XML elements, represent the public characteristics or attributes of a Grid Service, and OGSA defines it's own query mechanisms to obtain this information.

Exposing job ClassAds as individual service data elements enables external applications to query the scheduler about job queue information relying on OGSA query mechanisms. However a distinction must be made between the external representation of a ClassAd and the mechanisms required to update and maintain them. Even though we may expose ClassAds as service data, in order to allow remote queries, updates and persistent maintenance of ClassAds should only be achievable via adapted operations with required Condor semantics.

Asynchronous Notifications

Asynchronous notification in OGSA enables clients to register interest in being notified of particular messages. By exposing the scheduler as a notification source, we allow external applications to receive notification about job state related changes. 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. The asynchronous notification mechanism is very much interlinked with the service data concept - a notification being a ‘push’ of service data to subscribed clients, (i.e. clients are notified of modifications of service data elements of interest to them) - as such, this issue is very much related to exposing queue contents as previously described.

A potential source of concern is the potential 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. However this highlights the need for independent event services, providing reliable, ordered and client-specific event notification to be defined as part of the OGSI infrastructure.

Resource Claims

In order to service the jobs currently queued, the scheduler will claim a number of resources to which it was matched. The scheduler will attempt to reuse these claimed resources until these claims are revoked by the resource owner or the central manager. An interface allowing resource claims to be enumerated, inserted or obtained would be of great benefit. Apart from enabling users to verify available and claimed resources before submitting jobs, such an interface can allow resource requests to be made the scheduler, effectively enabling custom intra-job managers to interact with the rest of the system. This move away from a strict job model could be a first step towards a multi-layered approach to resource allocation and management on multiple pools across the Grid.
3.1.3 The shadow

The shadow, spawned by the scheduler once a resource has been allocated for a particular job execution request, acts as a resource manager for this request. It performs system calls - such as file I/O - on the submission site on behalf of the remotely-running application, handles checkpointing of the job, reschedules the job when halted on the assigned resource and monitors the execution site for job completion. It will update the corresponding job ClassAd with dynamic information during the running of the job (e.g. image size of the job), and will also execute “job policy” expressions embedded in the job ClassAd. These expressions enable users to specify a course of action to be taken when a particular condition is met, such as “kill the job after X hours”.

In addition, it is also responsible for acting as an ‘intra-job’ manager for parallel applications run on the Condor framework. PVM applications rely on the shadow to request new resources on which to spawn new process instances, providing they comply with a particular class of requirements. In turn, the shadow will rely on the scheduler to obtain resource allocations to satisfy these requests.

Exposing the shadow as a transient service instance would provide an abstract representation of a running job, providing an interface to control the execution of a job and resource management activities during execution time. This enables us, from a client perspective, to separate the process of requesting a job to be run from the management of the job during execution time. Apart from the benefits of a service virtualization of a running job, exposing the shadow could provide an interface to more complex intra-job resource management functions.

Lifetime management functions of Grid Service instances can be mapped to the lifetime of a job in the pool, and the shadow can act as a source of job-related notifications, specific to the running application. Submission of a job execution request to the scheduler would effectively correspond to a shadow instantiation request, the scheduler returning a Grid Service Handle (the global identifier of a Grid service) to the client once the shadow is generated. If implemented as persistent service the scheduler itself can be take on the role of a shadow factory. However the issue of transient factories, if the concept of transient schedulers is adopted, is still an open issue in the current Java OGSI implementation. We could however enable the scheduler to delegate the shadow instantiation request to a shadow-specific factory on the same host; a much preferable approach as it consequently enables us to deal with the synchronous implementation of factories in OGSI: 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. The GSH can then be exposed as service data and ‘pushed’ to the client through asynchronous notification. Furthermore a shadow-specific factory allows the scheduler to delegate a shadow instantiation to a factory on another host. This will considerably increase the scalability of the system by allowing load balancing when a large number of simultaneously running jobs must be managed.

If for any reason a job is halted on a remote resource, its corresponding shadow is responsible for rescheduling the job. In this particular case, we should take advantage of service activation and deactivation primitives provided by the OGSI
to ensure that the memory footprint of such a service is kept minimal during such periods of idleness.

3.1.4 The collector

The collector of a Condor pool is responsible for collecting information about daemons throughout a Condor pool, in the form of ClassAds sent periodically by other daemons in a pool such as resource ClassAds detailing resource characteristics, usage and job preferences, or scheduler ClassAds stating the fact that it has idle user jobs and should be contacted by the negotiator during a negotiation cycle.

Exposing the collector enables external applications to request information about the state of a pool and estimate resource availability in the Condor pool 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 Grid service, with XML representations of resource ClassAds exposed as individual service data elements. Though this service would be a persistent grid service for a ‘normal’ Condor pool, applications such as Condor glide-in, through which a pool can be built from temporarily assigned resources allocated via other resource management systems, could greatly benefit from the a transient collector instance. The lifetime of a collector instance would be mapped to the duration of a set of computations across such Grid resources.

3.1.5 The negotiator

The negotiator is responsible for enforcing pool-wide community policies. In this section we are not so much concerned with its matchmaking capabilities, which we will come back to when covering allocation services, but as a source of information regarding user priority and job preemption rules. The later is actually maintained by the accountant component of the negotiator, responsible for continually recalculating priorities based on current resource usage, in order to maintain fair-share usage principles.

The motivation behind exposing this service is to allow external applications to determine the policies by which resources will be allocated to service their job, complementing resource information obtained through the collector. If a client can verify the priority he is assigned in comparison to other users and determine the actions that will be taken if a higher priority user submits a job, it will enable him to more accurately determine the potential usage that can be obtained from the pool of resources.

The use of such a service would require a more suitable representation of customer capabilities to be adopted, encapsulating not only priorities but also authorization and resource access control information; the current association between name and floating point numbers requiring quite an intricate understanding of Condor’s allocation mechanisms. However the distributed nature of a Condor pool, and the fact that individual resource owners may determine their own policies make this an issue for future research outside of the scope of this document.
We must also consider that information such as job-preemption rules or user priorities could be published into the collector in order to be obtained remotely, rather than create an individual grid service for that purpose, essentially ensuring that the collector is the main resource information provider of the pool – or alternatively obtained via a separate accountant service.

The accountant component of the negotiator is responsible for maintaining and recalculating user priorities according to current usage and pre-assigned priorities. Creating an independent accountant service - through which user priorities can be obtained and usage recorded - would allow custom accountants to be developed, in order to replace Condor’s user priority computation mechanism with one suited to a specific environment.

3.2 Providing controlled access to Condor-managed resources

Whereas our first approach studied the use of OGSA to define 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.

As previously described, the motivations for this are to allow the entire set of Condor functionality to be exploited across multiple independently policed pools, by distinguishing resource management activities from job scheduling and execution management. Furthermore this would enable us to support Computing On Demand (CoD), a paradigm implemented in Condor allowing users to request job execution services directly from specific pre-allocated resources and enable access to server side components in a Condor glide-in setting, where Condor’s server-side daemons are submitted to and run by independent resource management systems via the GRAM, providing us with access to the resource once allocated. We must also consider “Grid Shell” developments, which improve the monitoring and reporting of a job running on a foreign Grid resource by creating a management wrapper for a job.

Another consequence of this distinction between resource management activities and job scheduling and execution management, is that these resource management activities would hence be capable of interfacing with a wider range of Grid Services with similar management requirements; providing Grid developers with an infrastructure on which to deploy services requiring CPU scavenging and other such high-throughput computing mechanisms.

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 using OGSA.

3.2.1 Flocking and Condor glide-in

The difficult problem of scheduling jobs and spreading Condor functionality across independently managed sites has been explored in Condor through Flocking, Condor-G and Condor glide-in.
Flocking allows a scheduler to submit a job to another pool if local resources cannot service it. The flocking mechanism is direct: the scheduler will interact directly with the central manager of another pool in order to request resources and access these as though accessing local pool resources. Condor glide-in on the other hand allows a distinction to be made between the resource allocation and job scheduling by submitting Condor’s server side daemons as jobs to resources managed by a foreign management system. When run, these will report back as available resources, enabling schedulers to bind a job to a resource only when its availability is determined.

It is based on these concepts that we define our current approach. By broadening the capabilities of Condor’s resource management services to incorporate OSGA services, we enable remote schedulers, who understand the temporal nature of allocated resources, to gain access to the full set of functionality provided by the latter, in a secure and controlled manner. We further consolidate these concepts by using OGSA standards to define an interface through which resource allocations can be requested, and increase the integratability of Condor by enabling a wider range of OGSA compliant clients and meta-schedulers to access Condor-managed resources across the Grid.

### 3.2.2 Condor resource management services

Resource management services in Condor operate according to high-throughput computing principles, allocating resources to users according to:

- **User requirements**: users submit a description of the execution environment required for their jobs.
- **Load balancing requirements**
- **Priorities and job pre-emption rules**: priorities can be assigned to users and job pre-emption rules can be defined to determine the course of action to take when high-priority users require resources in use lower-priority users.
- **Fair-share principles**: shares of resource usage are continuously evaluated to ensure that all users get a fair share of resources available based on their assigned priorities.

Resources are allocated on a ‘granting’ rather than sharing basis: resource owners grant access to their machine to a particular user (or more specifically a scheduler) until the agreement is revoked by the central manager or the resource owner. The scheduler will attempt to reuse this agreement with other idle jobs or hold it as a spare until it is revoked. A Condor pool is a highly dynamic environment: the owner or central manager may revoke allocation agreements at any time, and means to ensure a ‘graceful’ revocation of resources must be available.

To enable interaction of resource management services with OGSA Grid services, we need to determine how we can handle the following resource management activities performed by Condor using OGSA mechanisms:

- **Resource meta-data collection**: Resources in Condor advertise their characteristics by publishing resource usage information in the form of resource ClassAds through the collector.
- **Allocation request management:** schedulers advertise the fact that it has idle user jobs in the collector, in order to be contacted by the negotiator during a negotiation cycle. It is then that the negotiator will request information about the requirements for the next job a scheduler wishes to run on Condor resources.

- **Matchmaking:** the matchmaker is responsible for matching job ClassAds to resource ClassAds based on pool policies and contact both interested parties when a match is established, informing them of the identity of the opposite party. The ClassAd framework provides evaluation mechanisms that simplify the process of determining the correctness of a match based on the attribute list of the two ClassAds.

- **Claiming or allocation:** Upon being informed of a match, the client will begin the process of claiming the matched resource. The matched resource will verify that the network connection is from the actual matched client it was informed of by the matchmaker before accepting or rejecting the claim.

- **Revocation of allocation agreements**

The allocation agreements are provided through the pool’s central manager, whose role is essentially to permit or approve a potential matched allocation. It is left to the resource side components to actually accept this approval based on owner configuration. In this way Condor ensures that resource owner policies can be respected whilst enforcing community policies through the central manager and also compensates for the fact that matchmaking may be performed on potentially stale information about requirements or users, due to the highly dynamic nature of a pool.

We can hence identify 2 main sources of interaction for the client, which we will cover individually:

- The **central manager** will be responsible for providing an allocation interface through which clients can request the allocation of a resource meeting it’s requirements.

- Individual resources, once matched with a particular client, will grant the latter access to its services.

### 3.2.3 The central manager

Whilst in essence the role of the various condor components and their activities do not change, we must define their representation in order to achieve what has been outlined, and we focus here on the pool 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 on a regular basis the resource advertisements to the scheduler requirements and inform interested parties. This is very much suited to Condor’s asynchronous mode of operation.

The collector concept is not entirely dissimilar to that of an OGSA Service Group, or registry. A registry enables service discovery and provides mechanisms
to publish a Grid Service Handle (i.e. the global identifier of a service) and an
associated description in the form of XML elements based on soft state
registration. As registry contents are exposed as service data, users can then query
the registry by using the Grid Service query mechanisms.

At the heart of Condor’s matchmaking framework is the ClassAd. Classified
Advertisements (ClassAd) are essentially a property list: they provide mappings
between attributes and expressions. The ClassAd framework provides mechanisms
to establish the validity of the properties expressed in one ClassAd vis-à-vis that of
another. This is exploited in Condor through the matching of ClassAds
representing the user’s requirements when submitting a job, to resource ClassAds
stating the characteristics of the resources in a Condor pool. However the ClassAd
framework can be extended to embrace any kind of service and requirement that
may be described in the form of attribute/expressions associations.

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. This will also enable external applications to query the collector about
available resources and resource characteristics, before requesting allocations from
the pool.

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. From a client’s perspective the match will be in the
form of a resource GSH. It will also inform the corresponding startd of the match.

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.

### 3.2.4 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
slide-in, the condor_glidein program can be used to request the allocation of
resources from a specific site. When allocated and run on the remote resources,
Condor server-side executables will then report back to a collector running on the
client site. It is based on this information in the collector that jobs can subsequently
be matched to allocated resources.

### 3.2.5 Resource representation
Condor resource-side architecture maps quite well to OGSA’s factory concept. Resources are represented by a `startd` daemon, whose role is to register the resource with the collector and enforce local policies. Once mapped to a client by the negotiator, the `startd`, after mutual authentication with the client is successful, will spawn a starter daemon, whose role is to manage a specific job or set of jobs on behalf of the remote client. 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 to a client, and after successful mutual authentication, 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 Java 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. Once instantiated, the starter will forward all system calls performed by the running application to the shadow running on the client machine. The shadow – spawned by the scheduler to perform resource management activities for specific jobs- will then perform the requested calls on the local machine and forward the responses back to the starter. Support for this functionality, and other asynchronous communication between shadow, will require exposure of the shadow as a Grid Service instance. This ties in well with our previous description of how the shadow can be exposed, but in this case we are concerned with ‘internal’ communication with the shadow and its remote system call (RPC) functionality. The shadow is also responsible for deciding where job checkpoint files should be saved. Checkpointing allows a job to be resumed once it was halted on a remote machine by saving the state of the job during execution, if the job was re-linked using a Condor library. Rather than store checkpoint files on the submit machine, a centralized checkpoint server can also be used to ease the load on the submit machine. This checkpoint server could be exposed as an independent Grid service.

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.

We should also take note of the benefits in exposing the `startd/starter` combination when using Condor glide-in. The Condor glide-in strategy is a very important function of Condor, as it allows operation in an environment consisting of multiple, heterogeneous resource management systems. Whilst there is a lack of resource management systems capable of dealing with OGSA services, the daemons to be run on remote resources via the Globus (GT3) GRAM can be submitted with a self-contained OGSI hosting environment. 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. Further more, converting the starter to a Grid service also allows ‘Grid Shell’ functionality
to be exposed, which allows a job to be managed on a foreign resource and improves monitoring and reporting of job status information.

### 3.3 Security and Identity management

We consider the issue of identity management in this separate section, as it is a relatively vast and complex area, common to many services.

In order for us to exploit security mechanisms provided by the Grid Security Infrastructure (GSI) to handle identification of clients using x.509 Grid certificates, we must adopt a two-layered approach to authorization and access control: whilst GSI 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 it’s needs, specifying different roles (administrator/owner/negotiator/user) and levels of access (read/write) to components.

In cases where components do require access to an account local to the machine (strong authentication), such as the scheduler and shadow combination, we rely on GSI to provide credential mappings between Grid certificates and an account local to the submission site (on which the scheduler is deployed) under which invocations will be performed, and local files accessed - based on entries in a Grid map-file. It should be mentioned that in a grid setting, requiring remote users to have a local account on the resource they wish to access is perhaps not the most scalable solution and is costly administratively, even though it does considerably simplify accounting and policy management in heterogeneous environments. Several gateway technologies have been/are being developed to map remote users to temporary accounts. The incorporation of such mechanisms within the Condor framework is however beyond the scope of this document.

If we consider specifically the case of the scheduler, exposing the scheduler as a persistent grid service requires that credential mappings be performed for every service invocation. Allowing users to instantiate their own scheduler instances, however, would allow us to reap considerable security benefits. Whilst a Condor scheduler does not have to be run with root access, it is currently very much preferable; without root access, the scheduler can only spawn shadow instances under the same username it is running as; implying that in order for file i/o to be performed a user must grant read and write access to the scheduler and shadow combination. By allowing users to instantiate their own instances of the scheduler, we effectively eliminate the need for the scheduler itself to have root access. The responsibility of managing that privilege would be relegated instead to the scheduler factory.

However whilst the scheduler does effectively require a mapping to a local account for access to the local file system, 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, and its ability to operate without a shared file system. 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 \textit{username@domain} combinations. Condor components can currently be configured to use the previous version of GSI in this manner, as described in [15], and this capability should be updated where needed to incorporate the OGSI capabilities of GSI 3.

Means to share and convert grid-map file information between Condor services and sites in a pool whilst respecting individual site policies is an interesting area for future research.

### 3.4 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 \textit{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.
4. Evaluation and Conclusion

4.1 Evaluation

This report aims to identify alternatives by which OGSA principles and concepts could be exploited in the Condor framework, and hence attempts to be as broad as possible and consider the majority of Condor services. 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. We can assign different levels of immediacy to these architectural alternatives, in terms of desirability and accessibility of their implementation.

With these considerations in mind, the following would present the most interest:

- The scheduler and the collector

  The overall priority is to provide means for users to submit jobs to a Condor pool in an OGSA environment, with minimal interference to the overall intricate relationships between Condor components. Exposing the scheduler, in order to provide a submission and queue management interface, and the query API of the collector, in order to obtain information about the characteristics and availability of resources, should constitute the first step. The fact that these services have been made Web Service compliant clearly would ease their transitions to OGSA Grid Services.

  However, we have to decide the level of adoption of OGSA mechanisms within these components. Whereas we could expose the Condor specific API’s ‘as is’, it is clear that many benefits are to be gained in terms of integratability by introducing OGSA mechanisms to handle some elements of the functionality, with reference to exposing queue and collector contents as service data – allowing the querying to be performed via OGSA mechanisms, and be able to use notifications to inform customers of job state changes.

  Exposing the scheduler as a persistent Grid service would initially be sufficient to open Condor to the Grid. However this should be followed by the introduction of scheduling factories. On one hand, this ensures the removal of the need for the scheduler to have root access, and lets the responsibility of managing that privilege to be handled solely by the scheduler factory. On the other hand we allow the management of sets of jobs by users to be performed independently from each other on a same submission site. This would also allow for delegation of job management requests to schedulers local to a specific resource.

  Though exposing for example the shadow may be an interesting future development, any immediate implementation work should focus solely on the scheduler and collector.

- Startd and starter
The immediate appeal in individually exposing the services of these condor daemons (i.e. without necessarily implementing OGSA compliant resource management services) would be in allowing computing on demand functionality to be obtained directly from resources, and enabling interaction with the startd/starter when using Condor glide-in. Exposing Grid Shell functionality can also be of great interest, particularly to enable this individual functionality to be exploited by third party applications.

Though the overall priority should be given to the above, other architectural options may have implications that make them of interest for the Grid community. For example exposing resource management services would be a very interesting contribution. This does not necessarily imply that the entire set of services in a Condor pool have to be made OGSA-compliant: the focus would be on allowing third party developers to take advantage of Condor as a resource management platform on which to develop their own applications. There is a very strong demand for an open framework providing matchmaking, load balancing and CPU scavenging to be implemented for services on the Grid; and a lack of availability of brokering services capable of ClassAd-type matchmaking. Grid capable Condor resource management services would be perfectly suited for that matter. Providing for example a negotiator, collector accompanied with extensible stubs (or skeletons) for the client and server, will ensure that the basic resource management and allocation can be handled according to Condor principles.

### 4.2 Conclusion

We cannot deny that there are many differences in the overall concepts and mechanisms on which these technologies are built; the most crucial being that the synchronous implementation of Grid service invocations is not fully suited to the asynchronous mode of operation of Condor. Whilst the support for asynchronous notifications provided by OGSA does go some way towards bridging this gap, its focus on provision of content through a push of service data - which makes this content available to all authorized users – does not fully meet Condor’s requirements for reliable, ordered, and client-specific event message deliveries. An asynchronous implementation of the Open Grid Services Infrastructure could ultimately prove to be a useful contribution to the grid community, considering the nature of many Grid applications, based on heavy resource consumption and, hence, potentially long response times, and the fact that the Web Service standards on which OGSA is built support both synchronous and asynchronous communication. However there is still a need for a reliable event notification service to be defined independently from OGSA’s current service data notifications: OGSA’s approach to notification does have many advantages, as described when discussing how the queue contents of the scheduler can be exposed.

We must also consider the potential cost in resource usage that may be incurred. Incorporating an Open Grid Services Infrastructure built on Web Service hosting environments and commodity frameworks, and relying on XML-based standards – XML often considered to be a relatively heavy weight approach to data
representation – may appear to considerably increase the weight of our underlying Condor services. However this must be carefully balanced against the resource requirements of the applications run on the Condor framework, which may be significantly more important, and we must assume that this may vary considerably according to the choice of hosting environment. The resource implications of different OGSI hosting environments however are still being evaluated.

Despite of these issues, the conclusion of this report however is that many mechanisms introduced by OGSA can be mapped effectively into Condor’s architecture, and its usefulness as a standardized means to expose Condor as a set of Grid Services cannot be denied. By opting to embed OGSA considerations into the Condor architecture itself we ultimately ensure that new Grid concepts, advances and considerations that have been introduced by OGSA are embedded into the Condor architecture itself.

However, the acceptance of OGSA as the Grid standard is not yet established, whereas Condor is a well-established, mature middleware, which pre-dates the grid, but is increasingly taking center stage in many grid developments. Even though Grid standards are still immature, and will inevitably evolve rapidly, as demonstrated by the very recent announcement that OGSA will be decomposed into a family of standards, this is an important step towards making Condor an integral part of the Grid.
References

15. Heymann, E., Senar, M., Luque, E., Livny, M. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. Proceedings of