Resource Evaluation and Node Monitoring in Service Oriented Ad-hoc Grids

Ian Scriven[†] **Andrew Lewis**[†]
Matthew Smith[‡] **Thomas Friese**[‡]

[†]Griffith University
170 Kessels Road, Nathan 4111, Queensland
Ian.Scriven@student.griffith.edu.au A.Lewis@griffith.edu.au

[‡]University of Marburg
Philipps-Universität, Biegenstr. 10, 35032 Marburg, Germany
{matthew,thomas}@informatik.uni-marburg.de

Abstract

Ad-hoc grid computing is an emerging computing technology that promises to deliver high performance at relatively low cost using existing computing resources. There are a number of grid middleware systems being developed to this end. However, a number of features are lacking that are required if ad hoc grid computing is to become viable in a production environment. This paper addresses two of these key features – a resource evaluation and allocation system, which allows grid developers to accurately specify the requirements of their grid job to ensure the most suitable nodes are used when creating the ad-hoc grid, and a node monitoring and error recovery system, which allows grid applications to detect and recover from errors and complete successfully. These systems are built into Mage, the Marburg Ad-hoc Grid Environment, a grid middleware solution developed using the Globus Toolkit, Apache Tomcat and FreePastry.

Keywords: Grid Computing, Ad-hoc Grids, Peer-to-peer, Resource Evaluation, Node Monitoring.

1 Introduction

Grid computing is an increasingly popular alternative to traditional high performance computing systems. For many organizations, traditional supercomputers can be prohibitively expensive when the costs of administration and housing are combined with the initial purchase. An alternative is to use computing resources aggregated into a grid across an intranet or the Internet. Many such grids exist, or are developing. Most, however, have the drawback that to install and maintain production quality, large-scale grids is a complex and time-consuming task. It is to address this issue, and extend the potential benefits of grid computing to a wider audience, that ad hoc grids have been proposed.

The ad hoc grid computing paradigm brings together idle computing resources from varying geographical locations to form a one-off grid for a particular grid job. Once the job is completed, the grid is disbanded. This system has a number of advantages, including:

- Financial Control: Organizations can use idle CPU cycles from existing computers on their network to form a grid. This not only makes more efficient use of the resources already available but also allows for a gradual expansion of the grid without single large investments.

- Accessibility: Significant resources can made available to anyone on the network, as any connected node running the grid software is able to initiate a grid job. Of course, restrictions could be put in place if this freedom is not desirable.

- Flexibility: The grid can consist of heterogeneous systems running a variety of hardware and software configurations.

- Decentralization: As there is no central point of failure for the grid, the reliability of the grid is not affected by individual systems.

However, this system will not be suitable for some applications, where significant communication between grid nodes is required, as the connection between nodes will usually be low speed and high latency.

The heterogeneity and constantly changing nature of such a system also presents some unique challenges. The hardware and software configuration of nodes can differ vastly, and so some nodes may be better suited to particular jobs than others, and some may not be suitable at all. While allocation of work to nodes in an *existing* grid is a mature and well-defined field, there are no existing systems that can be used to evaluate and allocate nodes when forming a *new, ad hoc* grid. This paper will discuss a resource assessment and allocation mechanism that addresses these concerns, which allows the creation of ad hoc grids tailored to the specific needs of individual jobs.

Another set of challenges arises due to the fact that the nodes that make up the grid are not dedicated – the grid infrastructure is usually designed to run in the background while local users perform other tasks on the node (University of California 2007, Stanford University

2007), and at a lower priority than local tasks, which could cause starvation of the grid job. Nodes could also be turned off or restarted, or become otherwise unreachable (e.g. network problems). Grid applications must be able to detect and recover from these issues. This paper outlines an extensible node monitoring system that provides this capability.

The systems to be discussed are extensions of the Marburg Ad hoc Grid Environment (Mage) (Smith, Friese and Freisleben 2004). Mage is a WSRF compliant grid middleware solution, built upon the Globus Toolkit, Apache Tomcat and FreePastry, a peer-to-peer framework. It extends the Globus Toolkit, adding support for P2P based node discovery, service discovery and communication, along with non-intrusive service management (non-interrupting deployment and removal of services), intra-node resource isolation, and easy service development (using an Eclipse plug-in).

2 Resource Evaluation and Allocation

2.1 Overview of the Resource Evaluation and Allocation System

The resource evaluation and allocation (REA) system allows the grid application developer to accurately specify the requirements (and desirable qualities) that connected computers should display in order to be suitable for the grid job. The REA system provides this functionality through a series of steps including:

- Resource Request: The system allows users to specify node requirements for their grid job in the form of an XML file known as the resource request file. The format of this XML document is discussed in section 2.1.1 below.

- Node Discovery: In order to form an ad hoc grid, the initiating node must know what nodes exist on the network. Node discovery in Mage is performed using functionality provided by the FreePastry peer-to-peer system.

- Node Assessment: The system queries nodes discovered by the peer-to-peer system to determine whether they meet the mandatory requirements set out in the resource request. Information is obtained for desirable parameters once all mandatory parameters have been verified.

- Node Selection: The system returns a list of appropriate nodes sorted according to the user defined resource request file.

- Service Deployment: The resource allocation system can automatically deploy the grid service to the designated nodes, thus creating the ad-hoc grid. It can also facilitate the removal of the grid service upon completion of the job.

An overview of the communication that takes place between nodes during a call to the resource allocation system is provided in Figure 1. The arrows numbered one to three indicate major steps in the resource allocation process – node discovery, node evaluation and node selection. The dashed arrow represents two optional steps – retrieval of custom benchmarks and grid service deployment. Each step will be discussed in further detail in the following sections.

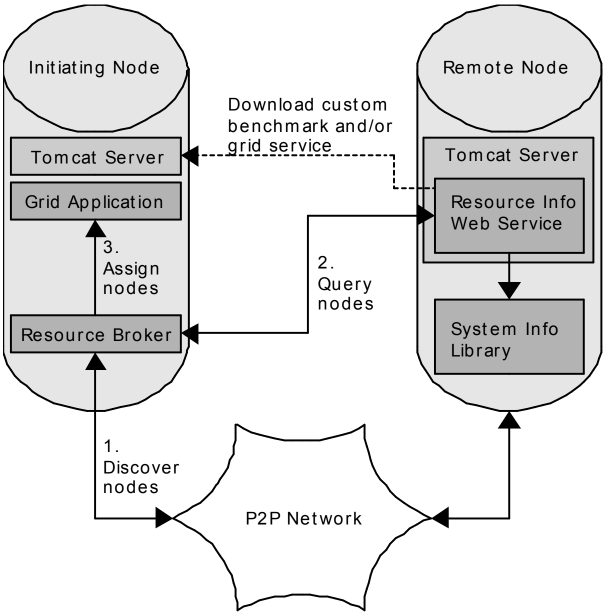

Figure 1: An overview of the resource allocation system

The resource allocation system presented here has a number of advantages over the Monitoring and Discovery System (MDS4) (Czajkowski, Fitzgerald, Foster and Kesselman 2001) provided by the Globus Toolkit. MDS4 uses one or more index services to act as centralized sources of information about available network resources. This introduces a central point of failure – problems with the node running the index service could potentially render a large number of nodes unavailable. Though this usually will not be a problem as dedicated, reliable systems are used to provide the indexing services, the Mage grid middleware software is designed to function in a purely peer-to-peer environment, where failure of one node will have no effect on the grid as a whole. As such, the REA system presented here performs node discovery and assessment through communication with the nodes themselves, rather than central index services. This process may be slower than using the MDS4, but it ensures all nodes available on the peer-to-peer network are considered when building an ad hoc grid. This resource allocation system also provides custom benchmarking functionality, which is not available in MDS4. Custom benchmarks, which will be discussed later in this paper, allow the resource evaluation and allocation system presented here to very accurately judge the suitability of nodes for a specific grid application, a significant advantage for applications having special requirements.

The REA system is used to make an ad hoc selection of nodes from a highly dynamic and heterogeneous environment, and from them construct a grid. Other systems proposed for resource allocation (Yang, Schopf and Foster 2003) are primarily targeted at distribution of tasks to a more or less static set of nodes to make most effective use of available resources under changing task loads. As such they have a different purpose to the system described in this paper. However, once an ad hoc grid has been constructed using this resource allocation system, the grid application could make use of other advanced scheduling algorithms, where past system load information is used to predict CPU load values, averages and variation over time. In the dynamic peer-to-peer grid environment, making use of gathered performance information (including results of benchmarks performed by the resource allocation system) when scheduling work could allow grid applications to more readily respond to changing resource availability.

2.1.1 Resource Request File

The resource request file is an XML document that is used to specify the requirements (both mandatory and desirable) of a grid job. A large number of parameters can be used, ensuring that the requirements of a grid job can be accurately specified. These parameters are CPU clock speed, the number of CPU cores available, the CPU architecture, the CPU manufacturer, the amount of RAM available, free disk space, the host operating system name and version, the java runtime environment version, system uptime, system CPU load, communication latency (i.e. ‘ping’ time), the availability of external program libraries, and finally the results of custom benchmarks, which will be discussed further in a later section.

Figure 2 shows an example resource request file. In this particular case, the job requires three nodes, and specifies a number of requirements that these nodes should adhere to. Firstly, the grid job requires nodes whose CPUs run at clock speeds higher than 1500 MHz. The rank tag here shows that nodes meeting all requirements will be sorted according to CPU speeds, and the three nodes with the highest CPU speeds will be selected for the grid job. The mandatory tag indicates that the selected nodes must satisfy this requirement. The next parameter is the node’s operating system name and version. The grid job requires nodes to be running Windows, version 5.0 (Windows 2000) or later. Finally, the availability of the software library LibraryA is checked (when dealing with Windows, this means that LibraryA.dll must be located in the appropriate directory).

2.1.2 Node Assessment

After the node discovery service has returned a list of nodes available on the network, the resource allocation system proceeds to query these nodes based on the requirements in the resource request. Node information is acquired through communication with a resource information web service that is available on all nodes running the grid infrastructure software. The resource information web service does not obtain system information directly, instead relying on another library, the system information library (SIL). Although the system information library currently supports only Windows NT and Linux based operating systems, the fact that it is separate from the rest of the resource allocation package means that developers could add support for other platforms by simply updating the system information library.

```
<resource-request>
   <nodes>3</nodes>
   <parameter type="cpu" mandatory="true">
      <value>1500</value>
      <modifier>orHigher</modifier>
      <rank>1</rank>
   </parameter>
   <parameter type=”os” mandatory=”true”>
      <name>Windows</name>
      <value>5.0</value>
      <modifier>orHigher</version>
   </parameter>
   <parameter type="library" mandatory="true">
      <name>LibraryA</name>
      <value>true</value>
   </parameter>
</resource-request>
```

Figure 2: A sample resource request file.

A graphical user interface has been created to allow the operator to quickly and easily create this resource request file. This interface also allows the user to inspect the results of the resource request once it has been completed.

The resource allocation system checks mandatory requirements first when querying nodes. If a node fails to meet a mandatory requirement, then it is removed from the list of possible nodes, and no further investigation of that node occurs. This minimizes unnecessary network traffic and increases the efficiency of the resource allocation system.

Currently, nodes are queried using direct communication with the resource information web service located on each node. While this may be satisfactory for smaller scale networks, it is not a particularly scalable approach. In environments where many nodes are present it would be preferable to utilize the peer-to-peer subsystem provided by Mage to efficiently multicast grid job requirements in order to reduce network load (especially on the initiating node). Nodes that were unsuitable for the grid job would simply not reply to the initial request.

2.1.3 Node Selection

Once all nodes have been investigated, the resource allocation system sorts them to provide the grid job with the best possible nodes. Nodes are sorted in accordance to ranking criteria provided in the resource request file (see Fig. 2). The required number of nodes can then be selected from the top of the list and returned for use in the grid job. Information on all nodes that met mandatory criteria is retained, however, in case more nodes are needed at some time during the job’s execution.

2.1.4 Service Deployment

Automatic and non-disruptive service deployment is critical to the viability of the grid (Smith, Friese and Freisleben 2004). Mage provides a hot deployment mechanism for this purpose. The resource allocation system makes use of this mechanism to provide automatic service deployment to remote nodes that are selected for a grid job. The grid service GAR file is delivered to the remote nodes over HTTP, making use of the web server component of Apache Tomcat, before being loaded on the remote node using the hot deployment mechanism. The resource allocation package uses the hot deployment mechanism again once the grid job has completed to remove the grid service from the remote node.

2.2 Custom Benchmarking

While the resource allocation system provides a wide range of inbuilt parameters, more information will often be needed when judging a node's suitability for a grid job. The resource allocation system allows for custom benchmarking classes to be created and used by grid developers to ensure that the best possible nodes were selected for the grid. One clear example of this is found in today's processor market. Clock speeds are no longer a valid way of comparing the performance of different CPUs – for example, an AMD Athlon64 3200+, which runs at a clock speed of 2 GHz, is comparable to Intel Pentium IV processors running at speeds of over 3 GHz. Custom benchmarks can be created to test integer or floating point operation speed, or memory access times, among other possibilities. For example, Figure 3 shows five nodes (with modern CPUs) as they would be sorted by the resource allocation system based on CPU clock speed. If the requesting grid job required three nodes based on clock speed, the three nodes running the Intel CPUs would be selected and used to create the ad-hoc grid. Depending on the individual grid job, this selection may not be optimal.

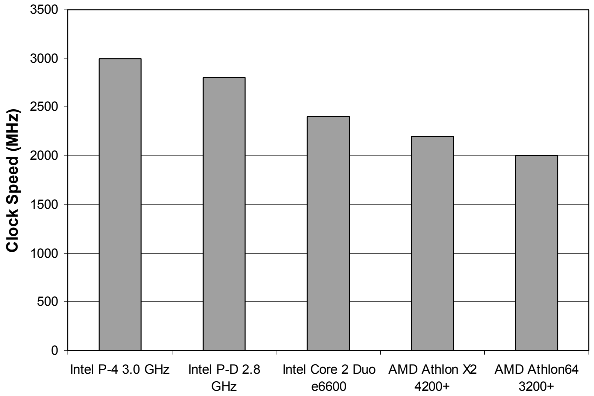

Figure 3: Various modern CPUs shown sorted by clock speed

Figure 4 shows the average results of a custom benchmark which performs a number of iterations of an optical design ray tracing algorithm. This benchmark, which utilises extensive floating point and trigonometric operations, shows that the two nodes with the highest CPU clock speeds (which were previously selected for use in the grid) are in fact the least suited of the five nodes for this type of computation. If an ad-hoc grid for an optical ray tracing problem was formed based on the results of this custom benchmark instead of the clock speed of the available nodes, the resultant grid would be much more suited to the job at hand and the problem could be solved faster with more efficient use of computing resources.

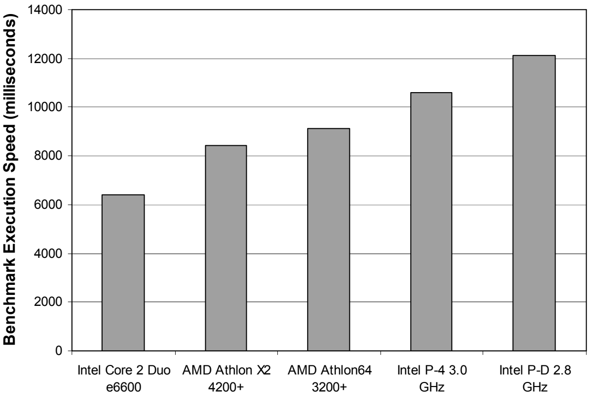

Figure 4: The same CPUs shown sorted by results of a custom benchmark (lower execution time is better)

Custom benchmarking can also prevent overloading of the 'best' nodes. As the workload of a node increases, its performance in custom benchmarks will degrade, making it appear less desirable to the resource allocation system.

The requirements for a custom benchmark are simple – it must be a single Java class file, having a main method (that is, it should be executable from a command line). Upon completion, it should output the result to standard output. The result must be an integer, which could represent the time in milliseconds the node took to complete some task, or different numbers could signify some other result, such as true or false. The value and modifier tags in the resource request file are used to signify what value, or range of values, is desirable (or mandatory) for the grid job.

Like grid service GAR files, custom benchmark classes are transferred to the remote nodes over HTTP using Apache Tomcat's web server component. In order to prevent abuse of this system, custom benchmarks are limited to one minute of runtime, and if necessary are forcibly terminated after this time period has elapsed.

3 Node Monitoring and Error Recovery

3.1 Overview of the Node Monitoring System

The ad hoc grid environment will usually not be as reliable as a dedicated high performance computing setup. The nodes that make up the ad hoc grid could be turned off or reset, the grid infrastructure software could be closed, the node could become unreachable, or the node may become heavily loaded by a local user. The node monitoring and error recovery system has been designed to address these problems, allowing grid programmers to detect these issues in their programs and react appropriately. The node monitoring and error recovery system is made up of a number of major components, including a server status grid service, a monitored service call, various monitor threads and an event handler, as shown in Figure 5. The roles of each of these components and the connections between them are described in the following sections of this paper.

3.2 The Monitored Service Call

Node monitoring is incorporated into a grid application using a monitored service call. The monitored service call uses a separate thread to call the grid service on the remote node. The service call thread is an abstract java class, and its run method must be implemented by the grid developer to make the required grid service call. While this call is being processed, the monitored service call monitors both communication with the remote node and the remote node's local CPU usage. Upon completion of the grid service call, the service call thread implementation must cast the results to an Object, before returning them using the finished method of the monitored service call. The monitored service call will then pass the results to the grid application, where they can be re-cast as the appropriate type. To facilitate this, the grid application must implement the returnResults method of the abstract service client class. The methods that deal with the results of the service call use the Object type because different grid applications will use a wide variety of types.

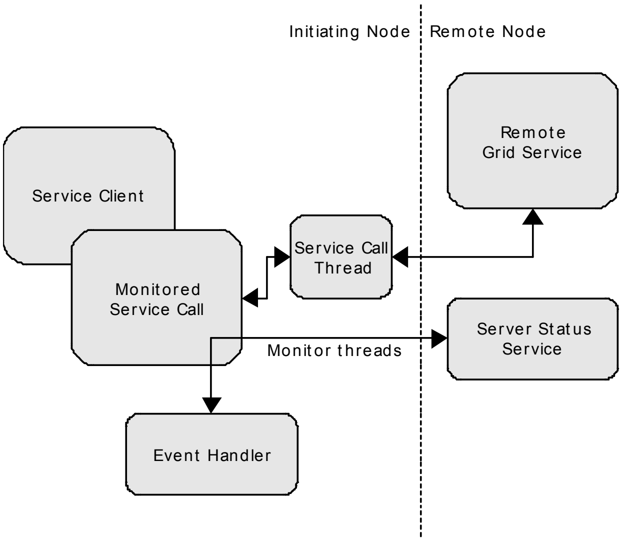

Figure 5: An overview of the node monitoring system

3.3 Node Monitoring

Nodes are monitored through calls to a server status grid service that is installed on all nodes. There are two monitor threads, one to check that the node is still reachable on the network (the communication monitor) and one to check the local CPU usage of the node (the load monitor). In the latter case, local CPU usage is assumed to be all CPU usage other than that of the grid infrastructure application (the modified Apache Tomcat server). Checks are performed at a default interval of twenty seconds, however this can be changed at any time. The communication monitor has a timeout of ten seconds – if there is no response from the remote node with this time, a communication error monitor event will be created. The load monitor can also produce a communication error if it times out, however it has a longer timeout of fifteen seconds to account for time needed by the server status service to accurately calculate the local CPU usage. If the local CPU usage exceeds a certain threshold (30% by default. However, this can also be changed.) a system load monitor event is produced. If the communication monitor or the load monitor completes a call without errors or warnings, a monitor event is produced to indicate this success.

3.4 Monitor Events

As described in the previous section, monitor events are produced as a result of the completion (successful or otherwise) of a communication or load monitoring call to the server status service on a grid node. A monitor event contains two fields – an integer code, which signifies what type of monitor event it is, and a message, which provides a human readable description of the monitor event.

3.5 Event Handling

The grid application uses an event handler to respond to monitor events. The event handler is an abstract class, and no implementations are provided. Different grid applications and environments will have vastly different requirements or limits imposed on resource use, and it is therefore critical that grid developers specify exactly how errors are to be handled and recovered from.

An event handler can be so simplistic as to simply echo any event messages to standard output on the initiating node, or as complex as to redistribute the workload around remaining nodes in the grid. If the previously mentioned resource allocation system is used, it could even allocate work to nodes that were not initially part of the ad hoc grid.

4 Implementation and Testing

A simple grid application was developed to test both the resource allocation system and the node monitoring system. This application uses an ad hoc grid to generate a Mandelbrot fractal pattern with different grid nodes constructing different sections of the image. The resource allocation system is used to construct the ad hoc grid, and the node monitoring system is used to ensure reliable completion of the job.

The application was run on a small test bed consisting of Intel Pentium III computers with clock speeds ranging from 450 MHz to 800MHz, running a mixture of Fedora Core 3 and 4. A simple resource request file was used to select the nodes with the highest CPU clock speeds for use in this job. The resource allocation system was demonstrated to correctly select and deliver appropriate resources to the application, either directly to specification or of the nearest available compromise.

Tests were also performed using the Mandelbrot application to assess the performance gains achievable through the use of the resource allocation system. Results of these tests are illustrated in Figure 6. A custom benchmark was created based on the Mandelbrot fractal generation algorithm to assess node suitability. The application was run a number of times on two different test bed environments. The first test involved machines with significantly different performance capabilities, and thus selecting the best possible nodes using the resource allocation system and custom benchmarking resulted in a significant decrease in computation time. The second test involved machines of more similar computational power, and as such the decrease in execution time was less pronounced. These tests show that in the heterogeneous ad hoc grid environment proper resource assessment and allocation can have a significant impact on overall grid performance.

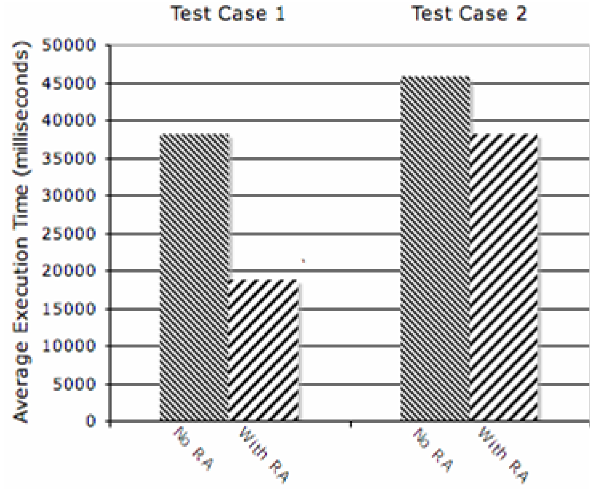

Figure 6: Testing the grid performance with and without resource allocation (RA). Test case 1 involved nodes with significantly different performance capabilities, while test case 2 used nodes with similar capabilities.

Multiple tests were also performed on the node monitoring system using different event handlers. The simplest test involved causing communication timeouts and node load warnings and having a basic event handler count the errors for each node and print the error information to standard output. Correct tracking of events was determined by inspection.

A second test involved using an event handler in conjunction with the resource allocation system to move work to a new node if a resource was deemed to have "failed". Failure was defined as two node communication errors occurring in a row, simulating either node failure causing lack of response, or network failure or congestion inhibiting access to a remote resource. During this test the combined functions of node monitoring and resource allocation allowed the application to complete successfully despite one of the grid nodes being removed from the network.

An issue that arose during testing was that when the event handler moved work from one node to another, there was no way to stop the original node from continuing to process the service call. This may occur if the action was in response to failure of network connectivity: the "failed" node would actually continue processing while being inaccessible. While this does not affect the grid application itself, as well as wasting computational capacity it will decrease perceived performance of the original node and any applications running on it. This is an area for future work.

5 Conclusions

In this paper two sub-systems extending the functionality of the Mage grid middleware system were discussed. A resource evaluation and allocation system allows ad hoc grids to be constructed using the best nodes available for a particular application. Grid job requirements can be accurately specified by a number of user-selected values for predefined parameters, or through the use of custom benchmarks. Once the required nodes are selected, the grid service can be automatically deployed to each node. A grid job can then immediately begin making calls to the grid service.

A node monitoring and error recovery framework allows grid jobs to run and complete successfully despite the volatile nature of the ad hoc grid. If a grid node becomes unreachable or heavily loaded, the work it was performing can be moved to a new node using the resource allocation system, or redistributed among remaining grid nodes, increasing grid resilience and fault tolerance.

With the framework in place, tools are planned to automate the process of incorporating the resource allocation and node monitoring and error recovery systems into grid applications in order to make the system as transparent to the grid application developer as possible. In particular, a more advanced error handler is planned that will allow the developer to configure its behaviour without any extra coding.

Both sub-systems have been implemented, integrated into the Mage system and functionally tested on a grid test bed. Work is currently ongoing to integrate the Mage system into Nimrod (Abramson, Sosic, Giddy and Hall 1995) and the Nimrod Portal in order to allow ad-hoc grids to be constructed and used in conjunction with traditional, fixed, high performance computing resources for large scale grid jobs. This would allow organisations to extend their pool of available high performance computing resources using existing desktop and personal computer installations.

6 References

Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C. (2001): Grid Information Services for Distributed Resource Sharing. *Proc. Tenth IEEE International Symposium on High-Performance Distributed Computing*, San Francisco, USA, 181-194, IEEE Computer Society.

Yang, L., Schopf, J.M. and Foster, I. (2003): Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments. *Proc. 2003 ACM/IEEE Conference on Supercomputing*, Phoenix, USA, 31-31, IEEE Computer Society.

Smith, M., Friese, T. and Freisleben, B. (2004): Towards a Service-Oriented Ad Hoc Grid. *Proc. 3rd International Symposium on Parallel and Distributed Computing*, Cork, Ireland, 201-208, IEEE Computer Society.

SETI@Home, University of California. http://setiathome.berkeley.edu/. Accessed 31/08/2007.

Folding@Home Distributed Computing, Stanford University. http://folding.stanford.edu/. Accessed 31/08/2007.

MAGE – The Marburg Ad-hoc Grid Environment, Smith, M., Friese, T. and Freisleben, B. http://mage.uni-marburg.de/. Accessed 02/09/2007.

The Globus Toolkit, The Globus Alliance. http://www.globus.org/toolkit/. Accessed 02/09/2007.

Apache Tomcat, The Apache Software Foundation. http://tomcat.apache.org/. Accessed 30/08/2007.

Pastry, Rice University. http://freepastry.rice.edu/. Accessed 01/09/2007.

Fedora Project, Red Hat, Inc. http://fedoraproject.org/. Accessed 02/09/2007.

GT Information Services: Monitoring and Discovery System (MDS), The Globus Alliance. http://www.globus.org/toolkit/mds/. Accessed 30/08/2007.

Abramson D., Sosic R., Giddy J. and Hall B. (1995): Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. *Proc. 4th IEEE Symposium on High Performance Distributed Computing*, Virginia, USA, 112-121, IEEE Computer Society.