Dynamic Resource Provisioning in Grid Environments

Ioan Raicu[1], Catalin Dumitrescu[1], and Ian Foster[1,2,3]

Abstract — Batch schedulers commonly used to manage access to parallel computing clusters are not typically configured to enable easy configuration of application-specific scheduling policies. In addition, their sophisticated scheduling algorithms can be relatively expensive to execute. Thus, for example, applications that require the rapid execution of many small tasks often do not perform well. Frey proposed that these problems be overcome by separating the two tasks of provisioning and scheduling. The provisioning component uses batch submissions (Condor "schedd" services in Frey's work) to acquire resources for application execution. The scheduling component dispatches application tasks to those resources. We introduce here a dynamic resource provisioning Web Services-based architecture, DRP, and use this architecture to evaluate new methods designed to optimize both the dynamic resource provisioning and task scheduling within dynamically provisioning resource sets. The task scheduling is implemented in a separate system DeeF, a distributed execution environment framework, which has DRP integrated to offer the necessary dynamic resource provisioning for generic execution of arbitrary codes on the DRP managed resources. Based on our performance evaluation, DRP can allocate resources in less than a minute after which a pool of resources can be maintained, increased, and decreased based on the load of the application using DRP. Our execution framework DeeF can process 100K fine granular tasks in 160 seconds; in such a high throughput workload, the overhead per task is 1.6 ms per task, a low enough cost that DeeF and DRP can enable a wide range of applications to run significantly more efficiently, due to efficient task dispatch and the dynamic resource provisioning that makes the compute resource management trivial from the application's viewpoint. We also address the performance implications various security mechanisms have on the execution framework and the dynamic resource provisioning.

Index Terms— dynamic resource provisioning, batch scheduler, interactive resource usage, Grid computing

1 Introduction

The increased scale of Grid [1] systems calls for minimizing the needs for human supervision and for automating and delegating as many management tasks as possible. We believe that the larger Grid systems get, the more important it will be to provide different scheduling policies for different applications and users.

Individual clusters in Grid environments such as TeraGrid [2] are typically managed by separate batch-queue schedulers, each of which implements a local space-sharing policy. A task submitted to a queue wait until suitable resources become available. If the request cannot be satisfied in its entirety, the time in queue can be variable and/or long—in many cases longer than the execution time itself.

Batch schedulers typically permit the configuration of queues that support a wide range of policies. Nevertheless, for any particular set of queue configurations, it will often be the case that 1) a user or application wants a different policy than is configured at a specific scheduler, and/or 2) the overhead of running the full queue processing on each job is high relative to the cost of executing the job. Even in an idle system (with zero queue wait time), complete scheduling (job submission, en-queuing, de-queuing, and starting of execution) can take 30~60 seconds, a relatively high cost for many small jobs.

The key issue we address in this paper is how to support the efficient execution of large numbers of small tasks in such batch scheduling environments. A partial solution to this problem is to configure a new queue with appropriate policy: e.g., to run all jobs from a certain user or application immediately. However, such reconfiguration can be difficult in practice, for administrative reasons. In addition, the need remains for each user task to be submitted to the batch scheduler.

An alternative approach, pioneered by Frey [3], avoids the need for reconfiguration of queues by submitting one or more requests for nodes and then deploying on those nodes software that implements a domain-specific scheduling strategy. By thus embedding a new scheduler inside the old, we separate the two tasks of resource *provisioning* (acquiring the resources needed for a computation) and *scheduling* (mapping tasks to those resources). In this way, we can address both of the concerns expressed in the second paragraph.

We can further refine resource provisioning by allowing the system to resize the set of resources it manages dynamically, for example by adding resources when demand is high, and releasing resources when demand is low. We term this method *dynamic resource provisioning*. Dynamic resource provisioning can be useful when either resource demand or resource availability varies during the course of program or workload execution, in which case we can improve execution times and/or the efficiency of resource utilization.

Resource provisioning methods have been explored by many researchers [3 - 9], as we review in Section 2 below. In our work, we seek to improve the performance of these methods by addressing both:

- *Efficient provisioning*: i.e., efficient strategies for acquiring and releasing resources; and

[1] Department of Computer Science, University of Chicago, IL, USA
[2] Computation Institute, University of Chicago & Argonne National Laboratory, USA
[3] Math & Computer Science Division, Argonne National Laboratory, Argonne IL, USA

- *Efficient scheduling*: i.e., efficient methods for dispatching requests to individual processors.

To this end, we define and implement an architecture that permits the embedding of different provisioning and scheduling strategies; implement a range of provisioning strategies, and evaluate their performance in some preliminary studies; and implement and evaluate the performance of a scheduling strategy. We show that our system performs better by factors of more than 10 for small tasks, relative to other dynamic provisioning approaches.

The rest of the paper is organized as follows: Section 2 covers the related work, Section 3 covers our dynamic resource provisioning architecture and performance evaluation, Section 4 covers the distributed execution environment framework and the performance evaluation, and Section 5 discusses the conclusions and future work.

2 Related Work

Frey and his colleagues pioneered the separation of provisioning and scheduling that we explore here via their work on Condor "glide-ins" in the late 1990s. The first published reference of which we are aware is Frey et al. [3]. Requests to a remote computer (submitted, for example, via Globus GRAM) are used to start "startd" services, which then register with a Condor resource manager that runs independently of the original batch scheduler.

Appleby et al. [4] were one of several groups to explore dynamic resource provisioning techniques within the context of a data center.

Singh et al. investigate, via simulations, the performance impact of resource provisioning on workflows [5]. Their simulations were performed using the Maui simulator over a workload trace collected from the NCSA TeraGrid cluster and 13 workflows. Their results show in general a reduction of about 50% in workflow completion time when using provisioning over no provisioning.

The MyCluster system [6] uses batch job submissions to deploy either Condor or Sun Grid Engine (SGE) services that then assemble into "personal clusters" running Condor or SGE.

Mehta et al. [7] use similar techniques to embed a Condor pool in a batch-scheduled cluster. However, they do not address dynamic behaviors, and have relatively high overhead when compared to our implementation.

Ramakrishnan et al. [8] also address resource provisioning. However, they make the simplifying assumption that resources are dedicated and available indefinitely, which is not the case in most batch-scheduled clusters. Their proposed solutions would have to be modified to be functional in most Grid systems.

Bresnahan et al. [9] describe a resource provisioning architecture specialized for the dynamic allocation of compute cluster bandwidth. A modified Globus GridFTP server supports the dynamic allocation of GridFTP data movers under server load. In contrast, our work supports the scheduling of any application.

3 Dynamic Resource Provisioning

As illustrated in Figure 1, our DRP system comprises: (1) user(s); (2) a DRP Utilizing application (in our case a Web Service); (3) the DRP Web Service; (4) a GRAM Web Service; and (5) a resource pool. The interaction between these various components is as follows.

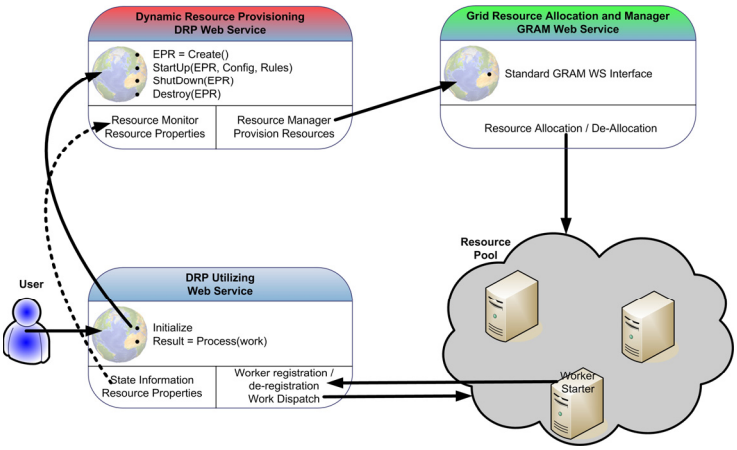

Figure 1: Dynamic Resource Provisioning Architecture

The DRP utilizing application initializes the DRP Web Service with a set of configuration parameters. These parameters include: the state that needs to be monitored and how to access it, the rule(s) and conditions under which DRP should allocate/de-allocate resources, the location of the worker code that is specific to the DRP utilizing application, the minimum/maximum number of resources it should allocate, the minimum/maximum length of time resources should be allocated for, and the allowed idle time per resource before resources are de-allocated. Once the DRP Web Service was initialized, the application would be ready to interact with its users and process work.

The users submit work to their application, which internally queues up the work making it ready for processing by a worker resource. The DRP Web Service monitors the internal queue state of the application, and based on the rules and conditions from the initialization phase, the DRP Web Service makes the decision how many resources and for how long to allocate. In our implementation, these resources are allocated using GRAM in order to abstract away all the local resource managers that could be used in Grids (PBS, LSF, Condor, etc). GRAM is used to bootstrap the worker starter code that is specific to the DRP utilizing application, which then registers with the application and becomes ready to process work. Once the application has workers available for work, it sends notifications directly to workers that work is available for pickup, after which the workers that received the notifications contact the application directly to pick up the relevant work.

While these worker resources are available (which is dictated by the resource de-allocation policies), any subsequent work requests from the user can simply use the same resources that have already been allocated (according to the resource allocation policy), without the need to go through the entire allocation process. We provide more details on the work dispatcher and worker starter in a more general sense in Section 4 when we discuss a generic distributed execution environment framework which takes the place of the DRP utilizing application from this section.

A specific instantiation of this architecture must specify a *resource acquisition* policy and a *resource release* policy, which we discuss in the next sub-section.

3.1 Resource Acquisition/Release Policies

A *resource acquisition policy* decides when and how to acquire new resources. This policy determines the state information that will be used to trigger new resource acquisitions (e.g., "if task queue length increases, acquire more resources"). It also determines the number of resources to acquire based on the appropriate state information, as well as the length of time for which the resources should be required. In our implementation of DRP, we rely on WS-GRAM [10] bundled with the Globus Toolkit 4 [11] for the coarse-grained resource allocation.

Having decided that n resources should be acquired, we then need to determine what request(s) to generate to the LRM to acquire those resources. We consider five different strategies in the work presented here.

The first strategy, **Optimal,** assumes that we can query the resource manager to determine the maximum number of resources available to us. We then simply request that number if it is less than n, and request n otherwise.

The other strategies assume that we cannot obtain this maximum number via a query. In the **One-at-a-time** strategy, we submit n requests for a single resource. In the **All-at-once** strategy, we issue a single request for n resources. In the **Additive**, strategy, for i=1, 2, …, the ith request requests i resources; thus, $\left\lfloor (\sqrt{8n+1}-1)/2 \right\rfloor$ requests are required to allocate n resources. Finally, in the **Exponential** strategy, for i=1, 2, …, the ith request requests 2^{i-1} resources. Thus, $\lceil \log_2(n+1) \rceil$ requests are required to allocate n resources.

Just as we have acquisition policies, we also need resource release policies. We distinguish between two classes of resource release policy (*centralized* and *distributed*), used to decide when to release already acquired resources.

In a **centralized policy**, decisions are made at a central location based on state information available at that location. For example: "if there are no tasks to process, release all resources," and "if the number of queued tasks is less than q, release a resource."

In a **distributed policy**, decisions are made at individual resources based on state information available at the resource. For example: "if the resource has been idle for time t, the acquired resource should release itself."

Note that resource acquisition and release policies are typically not independent: in most batch schedulers, one must release all resources obtained in a single request at once.

3.2 DRP Performance Evaluation

We use two metrics to evaluate our DRP system: *Provisioning Latency* (i.e., the time required to obtain all required resources) and *Accumulated CPU Time* (i.e., the total CPU time obtained since the first request to the DRP system). We expect these metrics to help us identify the best dynamic resource provisioning strategies in real world systems (i.e. TeraGrid).

We perform experiments in two scenarios on the ANL/UC TeraGrid site, which has 96 IA32 processors and is managed by the PBS local resource manager. In the first case, the site is relatively idle with only 2 of the 96 resources utilized; these results are shown in solid lines in Figure 2. Thus, our requests (for up to 48 resources) can be served "immediately." Due to PBS overheads, it takes about 40 seconds for the first resource to be allocated in all cases, despite the queue being idle; we observed this overhead vary between 30 seconds to as high as 100 seconds in other experiments we performed. Figure 2 shows the number of worker resources that have registered back at the application and are ready to receive work as the experiment time progressed; this time includes several steps: time to allocate the resources with GRAM, time needed to coordinate between GRAM and PBS the resource allocation, time PBS needed to prepare the physical resource for use, time needed to start up the worker code, and the time needed for the worker code to register back at the main application.

We see that the one-at-a-time strategy is the slowest, due to the high number of batch scheduler submissions: it takes 105 seconds to allocate all 48 resources vs. 22 to 36 seconds for the other strategies. Note that the accumulated CPU time after 3 minutes of the experiment for one-at-a-time is almost 30 CPU minutes behind the other strategies.

In a more realistic setting, sites are rarely idle, and hence some resource requests will end up waiting in the local resource manager's queue. To explore this case, we consider a scenario in which the site has only 47 resources available until the 160 second mark, at which point availability increases to 48; these results are shown in dotted lines in Figure 2. Thus, each strategy has their last resource request held in the wait queue until the 160 second mark. Those last requests are for 1, 3, 17, and 48 resources for One-at-a-time, Additive, Exponential, and All-at-once, respectively. On one extreme, the 1-at-a-time strategy manages to allocate 47 resources and has only 1 resource in the waiting queue; the other extreme, the all-at-once strategy has all 48 resources in the waiting queue waiting for a single resource to free up before it can process the entire request. This is evidence of the back-filling strategies of the local

resource manager. Therefore, the all-at-once is now the worst overall, being over 60 CPU minutes behind One-at-a-time and Exponential, and almost 90 CPU minutes behind Additive. Note that these lags in accumulated CPU time will remain until the resources begin to de-allocate, at which time the strategies that received their resources later will also hang on to the resources later; in the end, all strategies should get the same accumulated CPU time eventually.

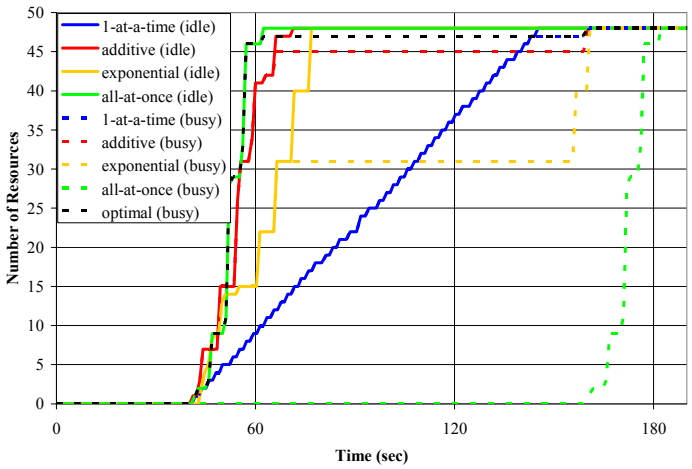

Figure 2: Provisioning latency in acquiring 48 resources for various strategies; the solid lines represent the time to acquire the resources in an idle system, while the dotted lines is the time to acquire the resources in a busy system

Table 1: Accumulated CPU time in seconds after 180 seconds in both an idle and busy system

Strategy	Accumulated CPU Time IDLE	Accumulated CPU Time BUSY
1-at-a-time	4220 sec	4205 sec
additive	6048 sec	5773 sec
exponential	5702 sec	4267 sec
all-at-once	6156 sec	409 sec
optimal	6059 sec	6059 sec

We conclude that different provisioning strategies must be used depending on how utilized a given set of resources are, with the all-at-once strategy being preferred if the resources are mostly idle, the additive and exponential strategies being appropriate for medium loaded resources, and the one-at-a-time being preferred when the resources are heavily loaded. Note that the finer grained the request sizes, the more likely it will be that DRP will be able to benefit from the back-filling of the local resource managers, but the higher the cost will be in terms of how fast the resources can be allocated. Notice

Another important issue, not addressed in this work, concerns the length of time for which resources should be requested. Many batch schedulers give preference to short requests and/or can schedule short requests into empty slots in their schedule (what is termed "backfilling"). Short requests may also minimize idle time. On the other hand, short requests increase more scheduling overhead and may cause problems for long-running user tasks. We envision the length of time to allocate resources to be application dependent, depending on the tasks complexity and granularity. Ideally, the length of time resources are allocated for should be large enough to ensure that several tasks can be performed on each resource, effectively amortizing the cost of the queue wait times for the coarse granular resource allocation.

4 A Distributed Execution Framework

DRP implements provisioning logic that maintains a pool of workers. To use DRP for a specific application, we must supply a worker implementation and a scheduler to dispatch tasks to workers.

We have integrated the DRP mechanism in an astronomy application called AstroPortal [12, 13] that performs the "stacking" analysis to the SDSS DR5 dataset. A user of the AstroPortal submits a stacking analysis which can be broken down into independent tasks (i.e. reading a region of interest [ROI] from the image data, calibration of the ROI, stackings of multiple ROIs, etc). These independent tasks are queued up in the AstroPortal, the DRP component allocates worker resources, and the worker resources process the tasks, which are later aggregate at the AstroPortal into the final result that is presented back to the user.

To show the suitability of DRP in a broader context, we have also developed a more general-purpose DRP-based distributed execution environment framework (DeeF), that allows the dispatch of arbitrary "tasks" to DRP-managed workers. We described DeeF in this section.

4.1 Architecture

As shown in Figure 3, DeeF comprises a *server* , the DeeF Web Service, which receives tasks from users, dispatches tasks to workers, collates results, and returns results to users, and a set of *workers* (created by DRP) that know how to execute tasks. Both the client interface to the server and all inter-component communication are Web Services (WS) based. The server runs within a Globus Toolkit 4 [11] container.

As shown in Figure 4, the server implements the classic factory/instance pattern, in which a factory service provides a *create resource* operation to allow a clean separation between multiple users of the service. To access DeeF, a user first requests creation of a new resource, to which is returned a unique endpoint reference (EPR) as a handle. The user can then use that EPR to submit tasks, monitor task progress, retrieve task results, and destroy the resource when it is no longer needed.

The worker code simply needs to register with the DeeF Web Service after which it will receive notifications of tasks that need to be processed, and where (which specific resource identified by the EPR) the tasks should be retrieved from.

The DeeF Web Service "submit tasks" interface takes as input an array of tasks, each specifying working directory, command to execute, arguments, and any required environment variables; it returns an array of outputs, each comprising the

original task that was run, the return code of the task, STDOUT contents, and STDERR contents.

The WSDL interface is easily extensible to support new features, such as user hints that can make the execution of tasks over DeeF more efficient. These hints could include information such as the names of data files that will be needed during task execution which could be used by a data caching mechanism, the expected execution time for each task, etc. We plan to integrate these hints in future revisions of DeeF, as well as other hints we find useful as we get DeeF and DRP integrated in other projects.

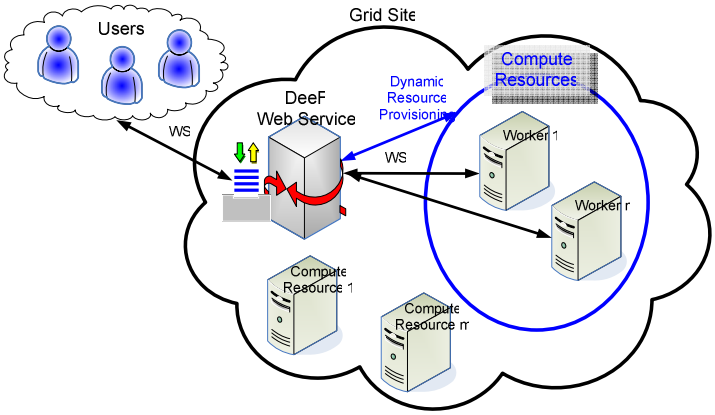

Figure 3: The DeeF architecture overview

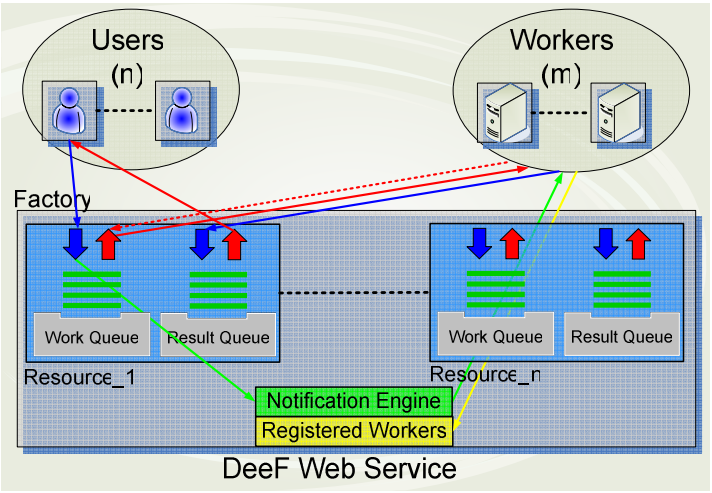

Figure 4: The DeeF Web Service implementation overview

Finally, as have built the DeeF and DRP prototypes using the Globus Toolkit, so we can easily leverage the Grid Security Infrastructure (GSI) to provide the various components of our systems transport-level security and message-level security; for each security level, there are three main options: authentication, integrity, and encryption of the data channels. The details of each of these security mechanisms are outside the scope of this paper, and we refer the readers to the Globus Toolkit 4 Grid security infrastructure overview [14].

4.2 DeeF Performance Evaluation

In order to test the overhead and efficiency of our execution environment, as well as the responsiveness of the DRP system, we executed a collection of 100K executables through DeeF without any security enabled and measured various metrics. The executable we ran was "/bin/echo." 100K "/bin/echo" executions required only 1.706 seconds on a single machine (2.4GHz Intel Xeon) when run from a local script.

We ran the same 100K executables through DeeF on the ANL/UC TeraGrid site with the following configuration: minimum number of resources set to 0, maximum number of resources to allocate was 48, DRP allocation strategy was all-at-once, and the ANL/UC site was relatively idle with more than 48 available resources. The user code had three main phases, which were overlapped in three different threads: sending tasks, processing tasks, and receiving task results. Figure 5 shows the throughput of the various phases and overlapping as time progresses.

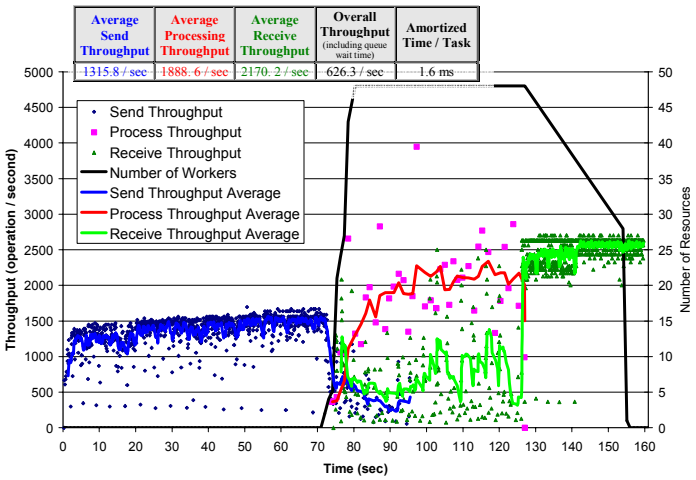

Figure 5: Performance evaluation of the DeeF framework with no security enabled

The 100K executables were completed in 159.75 seconds, which means the overall throughput was 626 tasks/sec, and the cost per task was 1.6 ms/task. This 1.6 ms/task is the overhead we expect to have when high throughput (and short lived) tasks are executed through DeeF. If a single task is submitted to DeeF that has workers already registered, the cost/task is about 200 ms, which is the upper bound we expect to have for any given task execution without security. With security enabled, the cost/task ranges from 750 ms to 1900 ms depending on the security technology used and level of security employed.

Finally, we wanted to compare our performance results directly with those of MyCluster, as it is the closest system at a functional level to our proposed DeeF system integrated with DRP. For our experiment, we configure DRP to allocate only one resource for the duration of the experiment; we then submit 30 tasks, each with a run time of 60 seconds, to DeeF. The total turnaround time, from time of first task submission to the time of last task completion, is then measured. We experimented with various secure and unsecure communication protocols in order to quantify the various overheads for our system, which can be used to directly compare our system's overhead with that of MyCluster.

Table 2: DeeF overhead for various levels of security

	Total Time (sec)	Overhead %
Tasks Execution	1800	0.00%
No Security	1806.16	0.34%
GSI Transport (Authentication)	1819.43	1.08%
GSI Transport (Authentication + Encryption)	1820.64	1.15%
GSI Secure Conversation (Authentication)	1825.56	1.42%
GSI Secure Conversation (Authentication + Encryption)	1829.38	1.63%
GSI Secure Message (Authentication)	1837.58	2.09%
GSI Secure Message (Authentication + Encryption)	1840.58	2.25%

Table 2 shows the summary of the DeeF overhead for the various levels of security. The execution time for DeeF to run this 30 task workload took 1806.16 seconds without any security. The ideal time with no overhead would have been 30*60 = 1800 seconds, so we computed our overhead to be 6.16 seconds or 0.34%. The most lightweight security mechanism we have available in GT4 is GSI transport security with authentication, which is the same level of security used in MyCluster. The MyCluster overhead ranged from 5% to 25% depending on which underlying scheduling technology they used (Condor or SGE respectively). We consider our over head to be substantially lower, with overheads ranging from 1.08% to 2.09% for the various authentication mechanisms. Notice that even with GSI secure message with encryption (GT4's most expensive security mechanism), we are still only at 2.25% overhead. It also interesting to note that encryption does not significantly increase the overhead for our simple example in which we have small messages being exchanged between the various components; we expect the encryption costs to increase if the input and/or output of each task becomes of significant size.

5 Conclusions and Future Work

Dynamic resource provisioning can lead to significant savings in end-to-end application execution time, enable the use of batch-scheduled Grids for interactive use, and alleviate the high queue wait times typically found in production clusters. We have described a dynamic resource provisioning architecture and presented preliminary performance results on the TeraGrid. We have also integrated DRP into DeeF, a distributed execution environment framework that would allow the execution of programs across a set of resources managed by DRP, and measured its performance which we presented in this paper.

We have integrated our DRP implementation with an astronomy application, AstroPortal [12, 13]. We have already begun the integration of the DRP functionality and DeeF into the Swift science and engineering workflow system [15] to enable workflows to decrease the ratio between queue wait times and the component execution time, which can be high when workflows are composed of many components. Finally, we plan on extending DRP to use multiple TeraGrid sites, allowing application that use DRP to transparently run across geographically distributed resources potentially harnessing hundreds to thousands of resources nationwide.

References

[1] I. Foster, C. Kesselman, S. Tuecke, "*The Anatomy of the Grid*", International Supercomputing Applications, 2001.

[2] TeraGrid, http://www.teragrid.org/

[3] J. Frey, T. Tannenbaum, I. Foster, M. Frey, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," Cluster Computing, vol. 5, pp. 237-246, 2002.

[4] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano - SLA Based Management of a Computing Utility," in 7th IFIP/IEEE International Symposium on Integrated Network Management, 2001.

[5] G. Singh, C. Kesselman and E. Deelman. "Performance Impact of Resource Provisioning on Workflows", ISI Technical Report 2006.

[6] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Creating Personal Adaptive Clusters for Managing Scientific Jobs in a Distributed Computing Environment", Workshop on Challenges of Large Applications in Distributed Environments, July 2006.

[7] G. Mehta, C. Kesselman, E. Deelman. "Dynamic Deployment of VO-specific Schedulers on Managed Resources," USC Information Sciences Institute, 2006.

[8] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi, J. Chase. Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control, IEEE/ACM SuperComputing 2006.

[9] J. Bresnahan, I. Foster. "An Architecture for Dynamic Allocation of Compute Cluster Bandwidth", MS Thesis, Department of Computer Science, University of Chicago, December 2006.

[10] M. Feller, I. Foster, and S. Martin. "GT4 GRAM: A Functionality and Performance Study", submitted to TeraGrid 07.

[11] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in IFIP International Conference on Network and Parallel Computing, 2005, pp. 2-13.

[12] I. Raicu, I. Foster, A. Szalay. "Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets", IEEE/ACM SuperComputing 2006.

[13] I. Raicu, I. Foster, A. Szalay, G. Turcu. "AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis", TeraGrid Conference 2006, June 2006.

[14] The Globus Security Team. Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective, Technical Report, Argonne National Laboratory, MCS, September 2005.

[15] I. Foster, J. Voeckler, M. Wilde, Y. Zhao. "Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation", SSDBM 2002.